Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation

An Object-Oriented Programming Model for Event-Based Actors

Diploma Thesis

Philipp Haller

May 2006

Referees:
Prof. Martin Odersky, EPF Lausanne
Prof. Gerhard Goos, Universität Karlsruhe

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Lausanne, den 15. Mai 2006

Abstract

Actors are concurrent processes which communicate through asynchronous message passing. Most existing actor languages and libraries implement actors using virtual machine or operating system threads. The resulting actor abstractions are rather heavyweight, both in terms of memory consumption and synchronization. Consequently, their systems are not suited for resource-constrained devices or highly concurrent systems. Actor systems that do provide lightweight actors, rely on special runtime system implementations.
Moreover, virtually all languages with a notion similar to actors support events or blocking operations only through inversion of control which leads to fragmentation of program logic and implicit control flow that is hard to track.
We show how lightweight actors can be implemented on standard, unmodified virtual machines, such as the Java Virtual Machine. For this purpose, we propose an event-based computation model which does not require inversion of control. The presented actor abstractions are implemented as a library for Scala rather than as language extensions.
The evaluation consists of two parts: In the first part we compare performance to an existing Java-based actor language. In the second part we report on experience implementing a distributed auction service as a case study.

Acknowledgments

First of all, I would like to thank Prof. Martin Odersky for welcoming me as a visiting student during the last six months. Without his constant support this thesis would not have been possible. Thanks to Dr. Lex Spoon for many helpful discussions and being a congenial office mate. Burak Emir provided numerous suggestions for improvement by correcting draft versions of this thesis. Thanks to Sébastien Noir for his help in finding and fixing bugs in the runtime system, and for porting it to JXTA. Prof. Jan Vitek provided valuable suggestions for improving the performance evaluation. Thanks to Dr. Sean McDirmid for making me think about the relationship between events and actors, and for pointing me to recent publications. Finally, I would like to thank Prof. Gerhard Goos for being my referee in Karlsruhe.

Contents

List of Figures

Chapter 1

Introduction

Concurrent programming is indispensable. On the one hand, distributed and mobile environments naturally involve concurrency. On the other hand, there is a general trend towards multi-core processors that are capable of running multiple threads in parallel.

With *actors* there exists a computation model which is especially suited for concurrent and distributed computations [HBS73, Agh86]. Actors are basically concurrent processes which communicate through *asynchronous message passing*. When combined with *pattern matching* for messages, actor-based process models have been proven to be very effective, as the success of Erlang documents [Arm96, NTK03].

Erlang [AVWW96] is a dynamically typed functional programming language designed for programming real-time control systems. Examples of such systems are telephone exchanges, network simulators and distributed resource controllers. In these systems very large numbers of concurrent processes can be active simultaneously. Moreover, it is very difficult to predict the number of processes and their memory requirements as they vary with time.

For the implementation of these processes, operating system threads and threads of virtual machines, such as the Java Virtual Machine [LY96], are usually too heavyweight. The main reasons are: (1) Over-provisioning of stacks leads to quick exhaustion of virtual address space and (2) locking mechanisms often lack suitable contention managers [DGV04]. Therefore,

Erlang implements concurrent processes by its own runtime system and not by the underlying operating system [Arm97].

Actor abstractions as lightweight as Erlang's processes have been unavailable on popular virtual machines so far. At the same time, standard virtual machines are becoming an increasingly important platform for exactly the same domain of applications in which Erlang–because of its process model–has been so successful: Real-time control systems [MBC+05, PFHV04].

Another domain where virtual machines are expected to become ubiquitous are applications running on mobile devices, such as cellular phones or personal digital assistants [Law02]. Usually, these devices are exposed to severe resource constraints. On such devices, only a few hundred kilobytes of memory is available to a virtual machine and applications.

This has important consequences: (1) A virtual machine for mobile devices usually offers only a restricted subset of the services of a common virtual machine for desktop or server computers. For example, the KVM [SMb] has no support for reflection (introspection) and serialization. (2) Programming abstractions used by applications have to be very lightweight to be useful. Again, thread-based concurrency abstractions are too heavyweight. Furthermore, programming models have to cope with the restricted set of services a mobile virtual machine provides.

A common alternative to programming with threads is, to use an event-driven programming model. Programming in explicitly event-driven models is very difficult [LC02].

Most programming models support event-driven programming only through *inversion of control*. Instead of calling blocking operations (e.g. for obtaining user input), a program merely registers its interest to be resumed on certain *events* (e.g. an event signaling a pressed button, or changed contents of a text field). In the process, *event handlers* are installed in the execution environment which are called when certain events occur. The program never calls these event handlers itself. Instead, the execution environment dispatches events to the installed handlers. Thus, control over the execution of program logic is "inverted".

Virtually all approaches based on inversion of control suffer from the following problems: (1) The interactive logic of a program is fragmented

across multiple event handlers (or classes, as in the state design pattern [GHJV95]), and (2) control flow among handlers is expressed implicitly through manipulation of shared state [CM06].

1.1 Goal

The goal of this thesis is to devise a programming model based on actors in the style of Erlang [AVWW96]. The programming model should be implemented as a library for *Scala*, a modern programming language which unifies functional and object-oriented programming [Oa04]. We want to adopt the following constraints:

1. All programming abstractions should be introduced as a library rather than by extending an existing language or inventing a new language. We believe that by using a modern programming language with general and well-defined constructs that work well together and provide the best of the object-oriented and functional programming worlds, domain specific languages, such as actor languages, can be implemented equally well as libraries.

2. Programming abstractions should not rely on special support from the underlying runtime environment. All library code should be runnable on unmodified popular virtual machines, such as the Java Virtual Machine (JVM) [LY96] and Microsoft's Common Language Runtime (CLR) [Gou02]. As these virtual machines neither provide means for explicit stack management, nor fine-grained control over thread scheduling, we will refer to them as *non-cooperative* in the following.

3. The implementation of our programming model should be suited for resource-constrained devices. Actor abstractions need to be memory efficient, and critical runtime services (e.g. serialization) have to be designed to be runnable on virtual machines for mobile devices. For example, certain configurations of virtual machines for embedded systems do not support reflection [SMb]. Therefore, target virtual machines cannot be assumed to provide general reflective serialization mechanisms.

4. We wish to support a very large number of simultaneously active actors. Targeting popular runtime environments, such as the JVM, this means that we cannot directly map actors to heavyweight virtual machine threads.

5. Our programming abstractions should be useful not only at the interface to other systems that support message passing. Rather, we wish to use actors as a general structuring abstraction that supports the composition of large, distributed systems utilizing modern multi-core processors. One could imagine e.g. a concurrent implementation of the Scala compiler in terms of actors.

1.2 Proposed Solution

To obtain very lightweight abstractions, we make actors *thread-less*. This poses a significant challenge as their execution state has to be saved and restored to support blocking operations. Moreover, virtual machines, such as the JVM, provide no means to explicitly manage the execution state of a program, mainly because of security considerations.

We overcome this problem by using *closures* as approximations for the continuation of an actor. By having our blocking operation *never return normally*, the continuation is even *exactly* defined by an appropriate closure. We can enforce this non-returning property at compile time through Scala's type system. At the same time, closures enable a very convenient programming style.

Although our implementation is event-based, it does not require inversion of control. Moreover, we achieve this without adding programming abstractions to cope with the modified computation model. The fact that the underlying computation is event-based is completely hidden from the programmer.

1.3 Contributions

The contributions of this thesis are three-fold:

1. We introduce *event-based actors* as an implementation technique for scalable actor abstractions on *non-cooperative* virtual machines.

   - To the best of our knowledge, event-based actors are the first to allow (1) reactive behavior to be expressed without inversion of control, and (2) unrestricted use of blocking operations, at the same time. Our actor library outperforms other state-of-the-art actor languages with respect to message passing speed and memory consumption by several orders of magnitude. Our implementation is able to make use of multi-core processors.
   - We show that by using Scala, actor primitives can be implemented as a library rather than as language extensions. Thus, our actor library may serve as an example for the implementation of domain specific languages in Scala.

2. By extending our event-based actors with a portable runtime system, we show how distributed Erlang [Wik94] can be implemented in Scala. Our library supports virtually all primitives and built-in-functions which are introduced in the Erlang book [AVWW96]. The portability of our runtime system is established by two working prototypes based on TCP and the JXTA[1] peer-to-peer framework, respectively.

3. We present the design and implementation of a state-of-the-art combinator library for type-safe serialization. The generated byte streams are compact, because of (1) structure sharing, and (2) base128 encoded integers. Our implementation is as efficient as Kennedy's [Ken04] without using circular programming [Bir84]. At the same time we support combinators which are more general than those of Elsman [Els04].

With our event-based actors we provide abstractions for concurrent programming based on asynchronous message passing which, arguably, make event-driven programming easier. More concretely, *explicit message passing* combined with *expressive pattern matching* allows a declarative programming style. Programmers can therefore concentrate on *what* to communicate instead of *how*. Furthermore, because our approach does not require *inversion of control*, we allow most high-level code be written in an intuitive, imperative thread-like style.

[1] http://www.jxta.org/

Chapter 2

Background & Related Work

2.1 Actor Model of Computation

Actors were introduced by Hewitt et al. [HBS73] and developed further by Agha [Agh86]. The actor model provides a self-contained unit that encapsulates both state and behavior. Communication between actors is only possible through asynchronous message passing. Upon arrival of a new message, an actor may react by

1. creating a new actor,
2. sending messages to known actors (messages may contain addresses of actors),
3. changing its own state.

An actor A may receive messages from any actor B that knows A's address. The order in which messages are received is unspecified. The original actor model [Agh86] requires message reception to be *complete*. Completeness guarantees that every sent message will be received after a finite duration. Particularly, no message will be lost. Ensuring completeness is hard and goes beyond the scope of our work. Instead, we adopt the "send and pray" semantics of Erlang [AVWW96], i.e. the system may fail to deliver a message at any time. We argue that this model is more practical,

especially in the context of mobile and (widely) distributed applications. Moreover, Erlang's success in the area of highly concurrent, distributed and fault-tolerant systems may justify the adoption of the semantics of its core abstractions [Arm96, NTK03].

2.2 Scala

Scala is a modern, statically typed programming language which unifies functional and object-oriented programming [Oa04]. It has been developed from 2001 in the programming methods laboratory at EPFL as part of a research effort to provide better language support for component software.

Scala code is compiled to run on the Java Virtual Machine [LY96] or Microsoft's Common Language Runtime [Gou02]. Existing libraries for these platforms can be reused and extended. Scala shares most of the type systems and control structures with Java and C#. Therefore, we restrict ourselves in the following to introduce concepts not found in those languages which are critical for understanding Scala code presented in this text.

2.2.1 Higher-Order Functions

Scala is a functional language in the sense that every function is a value. Thus, functions can be passed as parameters to, and returned from other functions.

For example, consider a function `forall` which tests if a given predicate holds for all elements of an array:

```
def forall[T](xs: Array[T], p: T => boolean) =
  !exists(xs, x: T => !p(x))
```

The type of the predicate `p` which is to be tested is the *function type* `T => boolean` which has as values all functions that take a value of type `T` as an argument and return a boolean value (note that `T` is a type parameter). Functional parameters are applied just like normal functions

(as in `p(x)`). Scala allows *anonymous functions* (i.e. functions which are not given a name) to be defined very concisely. In the example above, `x: T => !p(x)` defines an anonymous function which takes a value of type `T` and returns the negated boolean result returned by the application of `p`.

2.2.2 Case Classes and Pattern Matching

Scala allows *algebraic datatypes* to be defined using *case classes*. Case classes are normal classes tagged with the `case` modifier. Such classes automatically define a factory method with the same arguments as the constructor.

For example, algebraic terms consisting of numbers and a binary plus operation can be defined as follows:

```
abstract class Term
case class Num(x: int) extends Term
case class Plus(left: Term, right: Term) extends Term
```

Instances can be created by simply calling the constructors, as in

```
Plus(Num(1), Plus(Num(2), Num(3)))
```

Instances of case classes can be deconstructed using *pattern matching*. Case class constructors serve as elements of patterns.

For example,

```
def eval(term: Term): int =
  term match {
    case Num(x) => x
    case Plus(left, right) =>
           eval(left) + eval(right)
  }
```

evaluates algebraic terms.

In general, a matching expression

```
x match {
  case pat1 => e1
  case pat2 => e2
  ...
}
```

matches the value x against the patterns pat_1, pat_2, etc. in the given order. In the example, patterns are of the form `C(x1, ..., xn)` where `C` refers to a case class constructor and `xi` denotes a variable. A value matches such a pattern if it is an instance of the corresponding case class. In the process, the value is decomposed and its constituents are bound to variables. Finally, the corresponding right-hand-side is executed.

Variable patterns match any pattern and can be used to handle default cases. A variable pattern is a simple identifier which starts with a lower case letter.

2.2.3 Partial Functions

One of Scala's defining principles is that every function is a value. As every value is an object (because of Scala's unified object model), it follows that every function is an object. Therefore, function types are actually classes.

For example, a function of type `S => T` is an instance of the following abstract class:

```
abstract class Function1[-S, +T] { def apply(x: S): T }
```

The prefixes "-" and "+" are *variance annotations*, signifying contravariance and covariance, respectively. Functions with more than one argument are defined in an analogous way. Thus, functions are basically objects with `apply` methods.

As function types are classes they can be sub-classed. A very important subclass is *partial functions*. These functions are defined only in some part of their domain. Moreover, they provide a method `isDefinedAt` which tests whether it is defined for some value. In Scala's standard library, partial functions are defined like this:

```
trait PartialFunction[-A, +B] extends AnyRef with (A => B) {
  def isDefinedAt(x: A): Boolean
}
```

Instances of (anonymous) partial functions can be defined in a very concise way. Blocks appearing after a `match` expression are treated as instances of partial functions which are defined for every value that matches at least one of the specified patterns.

For example,

```
{
  case Incr() =>
    value = value + 1
  case Decr() =>
    value = value - 1
  case Reset() =>
    value = 0
}
```

defines a partial function which modifies a variable `value` when applied to instances of one of the case classes `Incr`, `Decr` or `Reset`.

2.3 Programming Actors in Scala

This section describes a Scala library that implements abstractions similar to processes in Erlang [AVWW96].

Actors [Agh86] are self-contained, logically active entities (in contrast, most objects in object-oriented systems are passive and become only active when a method is called) that communicate through asynchronous message passing. Each actor has a mailbox that can be manipulated only through the provided *send* and *receive* abstractions. Thus, the programming model is declarative and allows multiple flows of control.

In Scala, templates for actors with user-defined behavior are normal class definitions which extend the predefined `Actor` class. Figure 2.1 shows the

```
class Counter extends Actor {
  override def run: unit =
    loop(0)

  def loop(value: int): unit = {
    Console.println("Value: " + value)
    receive {
      case Incr() =>
        loop(value + 1)
      case Value(p) =>
        p ! value
        loop(value)
      case _ =>
        loop(value)
    }
  }
}
```

Figure 2.1: A simple counter actor.

definition of a simple counter actor. Concurrent behavior is specified analogous to threads in Java[1]: All actively executed code is contained in an overridden `run` method. Like a Java thread, an actor has to be started by calling its `start` method (inherited from `Actor`) which, in turn, triggers the execution of `run`.

For example,

```
val counter = new Counter
counter.start
```

creates a new instance of a counter actor and starts it.

By calling `receive` passing a list of *message patterns* with associated actions, an actor can remove messages from its mailbox and process them. The first message which matches one of the patterns is removed from the mailbox and the action corresponding to the first matching pattern is executed (the

[1] In fact, the `Actor` class defined in Scala's standard library extends `java.lang.Thread`. Although thread-less, our event-based implementation of actors we describe in chapter 3 provides essentially the same interface for compatibility.

order of message patterns is significant). If none of the messages in the mailbox can be matched against one of the patterns (or if the mailbox is empty), the call to `receive` blocks until an appropriate message can be processed. For example, a counter executing the `receive` statement given in figure 2.1 when its mailbox contains a single `Incr()` message, will remove the message from its mailbox and recursively call its `loop()` method passing an incremented counter value.

The mailbox of a freshly created actor is always empty. Messages can be added to an actor's mailbox only through its `send` method.

For example,

```
counter send Incr()
```

sends a `Incr()` message to our counter, thereby adding the message to its mailbox. If the counter was blocked, it is unblocked executing the action associated with `Incr()`. If it was not blocked (e.g. it was busy executing the action of a previous message reception), this simply means that the next call to `receive` will not block. Note that because message sends are non-blocking, the counter will execute its action *concurrently* with the continuation of the sender.

The message to be sent can have any subtype of `scala.AnyRef`, the supertype of all reference types. In contrast to channel-based programming [CS05] where a channel usually has to be (generically) instantiated with the types of messages it can handle, an actor can receive messages of any (reference) type.

To transparently support local as well as remote actors (i.e. running on a different node on the network), it is necessary to refer to actors using locality descriptors. Analogous to Erlang [AVWW96], each actor is uniquely identified by a *process identifier* (PID). An actor's PID can be obtained through its `self` property[2]:

```
val pid = counter.self
```

[2]In Scala accesses to fields and methods are *uniform*, i.e. they share the same syntax; thus, the implementation of a property can be changed from a field to (a pair of) accessor methods (for read and write access, respectively), and vice versa, without requiring changes to client code.

Sending a message to an actor through its PID is easy[3]:

```
pid ! Incr()
```

By sending its PID to other actors, an actor can become one of their *acquaintances*. For example, an actor interested in obtaining the value of our counter needs to send a message `Value(p)` where `p` is its PID[4]:

```
counter ! Value(self)
receive {
  case x: int => Console.println("Result: " + x)
}
```

The last **case** clause inside the counter's `receive` matches anything not matched by a preceding **case** clause. Thus, any message other than `Incr()` and `Value(p)` is ignored.

A method `receiveWithin` can be used to specify a time span in which a message should be received allowing an actor to timeout while waiting for a message. Upon timeout the action associated with a special `TIMEOUT()` pattern is fired. Timeouts can be used to suspend an actor, completely flush the mailbox, or to implement priority messages [AVWW96].

2.4 Actors for Smalltalk

Actalk implements actors as a library for Smalltalk-80 to study the actor paradigm of computation [Bri89]. Various actor computation models (e.g. Agha's actor model [Agh86], the communication protocol of ABCL/1 [YBS86]) are simulated by extending a minimal kernel of pure Smalltalk objects. No effort is made to use an event-based implementation; Instead, the concurrency model of the underlying environment is relied upon to be scalable. However, without extension, Smalltalk-80 does not support parallel execution of concurrent actors on multi-processors (or multi-core processors).

[3]Note that in Scala "!" is a valid name for a method and is not treated in any special way. Moreover, invocations of methods that take only a single parameter can be written infix. Thus, `counter ! Incr()` is short for `counter.!(Incr())`.

[4]The reader might be wondering how the type of `p` is determined as it is never mentioned inside the counter's class. Scala infers the type from the definition of `Value`.

ConcurrentSmalltalk [YT87] and Actra [TLD+89] pursue a deeper integration of actors into the Smalltalk environment. First, they show that Smalltalk needs to be extended in order to support concurrent actors on multi-processor machines. Actra extends the Smalltalk/V virtual machine with an object-based real-time kernel which provides lightweight processes. Thus, they rely on suitable support by the virtual machine. In contrast, we implement scalable actor abstractions on *non-cooperative* virtual machines.

2.5 Actor Foundry

The Actor Foundry is a class library for the construction of actor-based systems in Java[5]. As Java has no support for first-class functions or pattern matching (features typically found in functional programming languages), we do not expect the combination of Java together with such a library to be as expressive as typical actor languages or domain specific languages implemented using modern programming languages which combine functional and object-oriented programming, such as Scala. However, Scala's seamless interoperability with Java code makes it possible to build powerful abstractions using the classes and methods defined in the Actor Foundry. It has been shown that SALSA code (see 2.6) performs usually an order of magnitude better than Foundry code in Java, though [VA01].

2.6 SALSA

SALSA (Simple Actor Language, System and Architecture) [VA01] extends Java with concurrency constructs that directly support the notion of actors. A preprocessor translates SALSA programs into Java source code which in turn is linked to a custom-built actor library. The library provides actor classes which inherit from Java's thread class. Thus, SALSA suffers from the same scalability problems as general thread-based programming on the JVM. Moreover, the commitment of SALSA to language

[5]The Actor Foundry: A Java-based Actor Programming Environment, Open System Lab, 1998. See http://osl.cs.uiuc.edu/.

extensions contradicts our rationale for a library-based approach. However, as SALSA implements actors on the JVM, it is somewhat closer related to our work than Smalltalk-based actors or the Concurrency Control Runtime (see 2.7). Moreover, performance results have been published which enables us to compare our system with SALSA, using ports of existing benchmarks.

2.7 Concurrency Control Runtime

Chrysanthakopoulos and Singh [CS05] discuss the design and implementation of a channel-based asynchronous messaging library. Channels can be viewed as special state-less actors which have to be instantiated to indicate the types of messages they can receive. Usually, channels are used by composing join patterns (statically as well as dynamically), rather than by explicitly calling low-level operations. Similar to our approach, they implement channels as a library for C#. Instead of using heavyweight operating system threads they develop their own scheduler to support continuation passing style (CPS) code. Using CLU-style iterators blocking-style code is CPS-transformed by the C# compiler.

2.8 Timber

Timber [BCJ+02] is an object-oriented and functional programming language designed for real-time embedded systems. With respect to concurrency, it offers (1) a monitor-like construct for mutual exclusion, and (2) message passing primitives for both synchronous and asynchronous communication between concurrent *reactive objects*. The communication interface of reactive objects consists of methods which are executed as reactions to message sends. Methods can be asynchronous *actions* or synchronous *requests*. Invoking an action lets the sender continue immediately, thereby introducing concurrency.

Timber provides special support for programming real-time systems. Actions can be annotated with timing constraints (e.g. by specifying a deadline). A scheduler controls the order in which methods are executed try-

ing to satisfy all timing constraints. Thus, methods are not guaranteed to be executed in the order in which the corresponding messages are sent. However, if no timing constraints are given, messages are ordered according to their causal connections. If, in the sense of Lamport's "happened before" relation [Lam78], a message send must have happened before another send to the same object, the corresponding methods will be executed in the same order.

Timber is implemented as an extension to Haskell. The concurrent and stateful computations of reactive objects are based on monads. Thus, method bodies have to be written using Haskell's `do` notation.

Reactive objects cannot call operations that might block indefinitely. Instead, they install *call-back methods* in the computing environment which executes these operations on behalf of them. Completion of a blocking operation is typically signaled by an *event* that occurs inside the environment. Installed call-back methods serve as *event handlers* which are invoked by the environment. Consequently, their approach suffers from the usual problems with *inversion of control*: (1) The interactive logic of a Timber program is fragmented across multiple event handlers, and (2) control flow among handlers is expressed implicitly through manipulation of shared state.

2.9 Frugal Mobile Objects

Frugal objects [GGH+05] (FROBs) are distributed reactive objects that communicate through typed events. In response to a notification they can dynamically adapt their behavior to changes in the availability of resources, such as memory, bandwidth and CPU time. FROBs are programmed using a logically time-sliced computing model. In each time slice granted to a FROB by the runtime system, only a single, finite event handler is allowed to run. Event handlers are forbidden to use looping constructs of the underlying programming language. The prototype implementation of FROBs in Java cannot statically enforce this, though.

FROBs are basically actors with an event-based computation model, just as our event-based actors. The goals of FROBs and event-based actors are orthogonal, though. The former provide a *computing model* suited for

resource-constrained devices, whereas our approach offers a *programming model* (i.e. a convenient syntax) for event-based actors, such as FROBs. Currently, FROBs can only be programmed using a fairly low-level Java API (application programming interface) that directly maps concepts of the computing model to Java classes and methods. In the future, we plan to closely cooperate with the authors to integrate our two orthogonal approaches.

2.10 Java Extensions: JCilk, Responders

JCilk [DLL05] extends the Java language to support the passing of exceptions and return values from one thread to its "parent" thread that created it. It is a true semantic parallel extension of the base language, in the sense that its semantics is consistent with the existing semantics of Java's `try` and `catch` constructs. Our implementation of asynchronous exceptions (see 4.4.2) has been inspired by JCilk.

Recent work by Chin and Millstein [CM06] discusses a control-flow abstraction for an *event-loop* that avoids many of the drawbacks of inversion of control and the state design pattern [GHJV95]. We introduce a similar *event-loop* abstraction built on top of event-based actors which removes some of the limitations of their approach (see 3.6).

2.11 Summary

Most existing actor languages and libraries implement actors using VM or operating system threads. As a result, actors are rather heavy-weight, both in terms of memory consumption and synchronization. Consequently, their systems are not suited for resource-constrained devices or highly concurrent systems. Actor systems that do provide light-weight actors, such as Erlang or Actra, rely on special runtime system implementations. In contrast, we show how light-weight actors can be implemented on standard, unmodified virtual machines, such as the JVM. Finally, virtually all languages with a notion similar to actors support event-driven programming only through inversion of control which leads to fragmentation of

program logic and implicit control flow that is hard to track. Our contribution is an implementation technique for actors that supports event-driven programming avoiding inversion of control.

Chapter 3

Event-Based Actors

Logically, an actor is not bound to a thread of execution. Nevertheless, virtually all implementations of actor models associate a separate thread or even an operating system process with each actor [Bri89, TLD+89, BG99, VA01].

In Scala, thread abstractions of the standard library are mapped onto the thread model and implementation of the corresponding target platform, which at the moment consists of the Java Virtual Machine (JVM) [LY96] and Microsoft's Common Language Runtime (CLR) [Gou02].

An approach where each actor is assigned its own thread does not scale on either of the two platforms because the respective thread model is too heavyweight. The main reasons are: (1) over-provisioning of stacks which leads to quick exhaustion of virtual address space (at least on 32-bit machines) and (2) locking mechanisms which lack suitable contention managers [DGV04].

To overcome the resulting problems with scalability, we propose an event-based implementation. The event-based character of our implementation stems from the fact that (1) actors are thread-less, and (2) computations between two events are allowed to run to completion. An event in our library corresponds to the arrival of a new message in an actor's mailbox.

The rest of this chapter is structured as follows. First, by means of an example, we point out the challenges an event-based implementation of

actors has to face. Thereby, the central ideas of our proposed solution are explained intuitively. In section 3.2 we show how concurrent actors can be executed on a single thread. Extensions for multi-processors and multi-core processors are discussed in section 3.3. In section 3.4 we describe a scheduler for actors which guarantees progress even in the presence of blocking operations. Finally, applications of actors to event-based programming are discussed in section 3.6.

3.1 Execution Example

First, we want to give an intuitive explanation of how our event-based implementation works. For this purpose, consider the execution of two actors A and B:

A:

```
...
B ! Value(7)
...
```

B:

```
...
receive {
  case Value(x) =>
    receive {
      case Value(y) =>
        Console.println(x + y)
    }
}
```

Because actors and threads are decoupled, for the sake of simplicity, assume that both actors are running on the same thread and that A's send statement gets executed first (note that in Scala, methods that take only one argument can be written infix). Send appends message `Value(7)` to B's mailbox. Because the arrival of a new message might enable the target actor to continue, send will transfer control to B. Thus, the `receive` statement of actor B is executed on the sender's thread. According to the

semantics of `receive`, the new message is selected and removed from the mailbox because it matches the first (and only) case of the outer `receive`. Then, the corresponding action is executed with the pattern variables bound to the constituents of the matched message (i.e. $x = 7$). The block enclosed in curly braces following the outer `receive` actually defines a partial function. Thus, executing the corresponding action is done by applying the partial function to the matched message:

```
{
  case Value(x) =>
    receive {
      case Value(y) =>
        Console.println(x + y)
    }
}.apply(Value(7))
```

Intuitively, this reduces to

```
receive {
  case Value(y) => Console.println(x + y)
}
```

where x is bound to the value 7.

Assuming there is no other message in the actor's mailbox, logically, this call to `receive` blocks. Remember that we are still inside the call to `send` (i.e. send did not return yet). Thus, blocking the current thread (e.g by issuing a call to `wait()`) would also block the call to send.

This is illegal because in our programming model `send` has a non-blocking semantics. Instead, we need to suspend B in a way that allows `send` to return. For this, inside the (logically) blocking `receive`, first, we remember the rest of the computation of B. In this case, it suffices to save the closure of

```
receive {
  case Value(y) => Console.println(x + y)
}
```

Second, to let send return, we need to unroll the runtime stack up to the point where control was transferred to actor B. We do this by throwing a

special exception inside the blocking `receive` which gets caught in `send`. After catching the exception `send` returns normally. Thus, `send` keeps its non-blocking semantics.

In general, though, it is not sufficient to save a closure to capture the rest of the computation of an actor. For example, consider an actor executing the following statements:

```
val x = receive {
  case y: int =>
    f(y)
}
g(x)
```

Here, `receive` produces a value which is then passed to a function. Assume `receive` blocks. Remember that we need to save the rest of the computation *inside* the blocking `receive`.

To save information about statements following `receive`, we would either need to (a) save the call-stack, or (b) rely on support for *first-class continuations*.

Virtual machines like the JVM or Microsoft's CLR provide no means for explicit stack management, mainly because of security reasons. Thus, languages implementing first-class continuations have to simulate the run-time stack on the heap which poses serious performance problems [BSS04]. Moreover, programming tools such as debuggers and profilers rely on run-time information on the native VM stack which they are unable to find if the stack that programs are using is allocated on the heap. Consequently, existing tools cannot be used with programs compiled using a heap-allocated stack.

Thus, most ports of languages with continuation support (e.g. Scheme [KCR98], Ruby [Mat02]) onto non-cooperative virtual machines abandon first-class continuations altogether (e.g. JScheme [AHN], JRuby[1]). Scala does not support first-class continuations, primarily because of compatibility and interoperability issues with existing Java code.

To conclude, both approaches for managing information about statements following a call to `receive` would require changes either to the compiler

[1]See http://jruby.sourceforge.net.

or the VM. Following our rationale for a library-based approach, we want to avoid those changes.

Instead, we require that `receive` *never returns normally*. Thus, managing information about succeeding statements is unnecessary.

Moreover, we can enforce this "no-return" property at compile time through Scala's type system which states that statements following calls to functions (or methods) with return type `scala.All` will never get executed ("dead code") [Oa04]. Note that returning by throwing an exception is still possible. In fact, as already mentioned above, our implementation of `receive` relies on it.

3.2 Single-Threaded Actors

Before diving into the implementation of our two communication abstractions we want to point out another issue. As we want to avoid inversion of control `receive` will (conceptually) be executed at the expense of the sender. If all actors are running on a single thread, sending a message to an actor A will re-evaluate the call to `receive` which caused A to suspend.

A simplified implementation of send for actors running on a single thread looks like this:

```
def send(msg: Message): unit = {
  sent += msg
  if (continuation != null && continuation.isDefinedAt(msg))
    try {
      receive(continuation) // saves continuation as side-effect
    } catch {
      case Done =>
        // continuation already saved
    }
}
```

The sent message is appended to the mailbox of the actor which is the target of the `send`. Let A denote the target actor. If the `continuation` attribute is set to a non-null value then A is suspended waiting for an appropriate

message (otherwise, A did not execute a call to `receive`, yet).

`continuation` refers to (the closure of) the partial function with which the last blocking `receive` was called. Thus, we can test if the newly appended message allows A to continue (trait `PartialFunction[-A,+B]`, which inherits from the function type `(A => B)`, defines a method `isDefinedAt(x: A): Boolean`).

Note that if, instead, we would save `receive(f)` as continuation for a blocking `receive(f)` we would not be able to test this but rather had to blindly call the continuation. If the newly appended message would not match any of the defined patterns `receive` would go through all messages in the mailbox again trying to find the first matching message. Of course, the attempt would be in vain as only the newly appended message could have enabled A to continue.

If A is able to process the newly arrived message we let A continue until it executes a blocking, nested `receive` or finishes its computation. In the former case we need to make sure that `send` is not blocked but, instead, can return normally because of its non-blocking semantics.

By catching a special exception of type `Done` which is thrown by the blocking, nested `receive` (see below), `send` can pretend not having executed A at all.

Technically, this trick unrolls the call-stack up to the point where `send` transferred control to A. Thus, to complete the explanation of how the implementation for `send` works, we need to dive into the implementation of `receive`.

3.2.1 Receive

`receive` selects messages from an actor's mailbox and is responsible for saving the continuation as well as abandoning the evaluation context:

```
def receive(f: PartialFunction[Message, unit]): scala.All = {
  sent.dequeueFirst(f.isDefinedAt) match {
    case Some(msg) =>
      continuation = null
```

```
        f(msg)
        die()
      case None =>
        continuation = f
    }
    throw new Done
  }
```

Naturally, we dequeue the first message in our mailbox which matches one of the cases defined by the partial function which is provided as an argument to `receive`. Note that `f.isDefinedAt` has type `(Message => boolean)`. As the type of the resulting object is `Option[Message]` which has two cases defined, we can select between these cases using pattern matching [Oa04]. When there was a message dequeued we first reset the saved continuation. This is necessary to prevent a former continuation to be called multiple times when there is a send to the current actor inside the call `f(msg)`. After message processing `die()` sets a flag which indicates that the execution of the actor has terminated. When an actor has died, it is not able to receive messages any more, thus preventing its mailbox from being flooded.

If we didn't find a matching message in the mailbox, we remember the continuation which is the closure of `f`. In both cases we need to abandon the evaluation context by throwing a special exception of type `Done`, so the sender which originated the call to `receive` can continue normally (see above).

3.3 Multi-Processors and Multi-Core Processors

To leverage the increasingly important class of multi-core processors we want to execute concurrent activities on multiple threads. We rely on modern virtual machine implementations to execute concurrent VM threads on multiple processor cores in parallel.

A scheduler decides how many threads to spend for a given workload of concurrent actors, and, naturally, implements a specific scheduling strategy. Because of its asychronous nature, a call to `send` introduces a concurrent activity, namely the resumption of a previously suspended actor. We

encapsulate this activity in a *task item* which gets submitted to the scheduler (in a sense this is a *rescheduling send* [SS02]):

```
def send(msg: Message): unit = synchronized {
  if (continuation != null
      && continuation.isDefinedAt(msg)
      && !scheduled) {
    scheduled = true
    Scheduler.putTask(new ReceiverTask(this, msg))
  }
  else
    sent += msg
}
```

If a call to send finds the current continuation of the receiving actor A to be undefined, A is not waiting for a message. Usually, this is the case when a task for A has been scheduled that has not been executed, yet. Basically, send appends the argument message to the mailbox unless the receiving actor is waiting for a message and is able to process the argument message. In this case, we schedule the continuation of the receiving actor for execution by submitting a new task item to the scheduler.

The scheduler maintains a pool of worker threads which execute task items of type `ReceiverTask`. A `ReceiverTask` is basically a Java `java.lang.Runnable` that receives a specified message and has an exception handler that handles requests for abandoning the evaluation context:

```
class ReceiverTask(actor: MailBox, msg: MailBox#Message)
      extends Runnable {
  def run(): unit = {
    try {
      actor receiveMsg msg
    }
    catch {
      case Done =>
        // do nothing
    }
  }
}
```

`receiveMsg` is a special form of `receive` which processes a given message

according to the actor's continuation. This is needed to preserve the semantics of `receive` in a multi-threaded setting as between scheduling and execution of a task item more messages could be sent to the receiving actor; thus, the first matching message in the mailbox and the message that triggered the execution might be different at the time when the task item is executed.

Actors are not prevented from calling operations which can block indefinitely. In the following we describe a scheduler which guarantees progress even in the presence of blocking operations.

3.4 Blocking Operations

The event-based character of our implementation stems from the fact that (1) actors are thread-less, and (2) computations between the reception of two messages are allowed to run to completion. The second property is common for event-driven systems [HSW+00] and reflects our assumption of a rather interactive character for most actors. Consequently, computations between arrival of messages are expected to be rather short compared to the communication overhead.

Nevertheless, we also want to support long-running, CPU-bound actors. Such actors should not prevent other actors from making progress.

Likewise, it would be unfortunate if a single blocking actor could cause the whole application to stop responding, thereby hindering other actors to make progress.

We face the same problems as user-level thread libraries: Processes yield control to the scheduler only at certain program points. In-between they cannot be prevented from calling blocking operations or executing infinite loops. For example, an actor might call a native method which issues a blocking system call.

In our case, the scheduler is executed only when sending a message leads to the resumption of another actor. Because send is not allowed to block, the receiver (which is resumed) needs to be executed on a different thread. This way, the sender is not blocked even if the receiver executes a blocking

operation.

As the scheduler might not have an idle worker thread available (because all of them are blocked), it needs to create new worker threads as needed. However, if there is at least one worker thread runnable (i.e. busy executing an actor), we do not create a new thread. This is to prevent the creation of too many threads even in the absence of blocking operations.

Actors are still thread-less, though: Each time an actor is suspended because of a blocking (which means unsuccessful) receive, instead of blocking the thread, it is *detached* from its thread. The thread now becomes idle, because it has finished executing a receiver task item. It will ask the scheduler for more work. Thereby, threads are reused for the execution of multiple actors.

Using this method, an actor-based application with low concurrency can be executed by as few as two threads, regardless of the number of simultaneously active actors.

The first picture in figure 3.1 shows an example for the execution of three actors on two worker threads. The two columns represent two worker threads. Time increases downwards. Arrows between columns represent message sends. A line ending with a horizontal bar means an actor is suspended because of an unsuccessful `receive`. Upon an appropriate send, execution of an actor can be resumed on any idle worker thread. For example, actor A is executed on two different threads during its lifetime.

The second picture shows a situation where the scheduler has to create a new worker thread because of blocked worker threads. First, actor A sends a message to B. Both actors run concurrently, until B issues a blocking operation, indicated by the cross. Thus, when A sends a message to C, thereby resuming it, all worker threads other than the thread executing A are blocked. Because C might call blocking operations, it is not allowed to be executed on the same thread as A. Therefore, the scheduler creates a new worker thread, indicated by the third column, which starts executing C. Meanwhile, B is unblocked and finishes its execution (up until the next unsuccessful `receive`). Thus, when C sends its message, A can be resumed on the worker thread that was used to execute B.

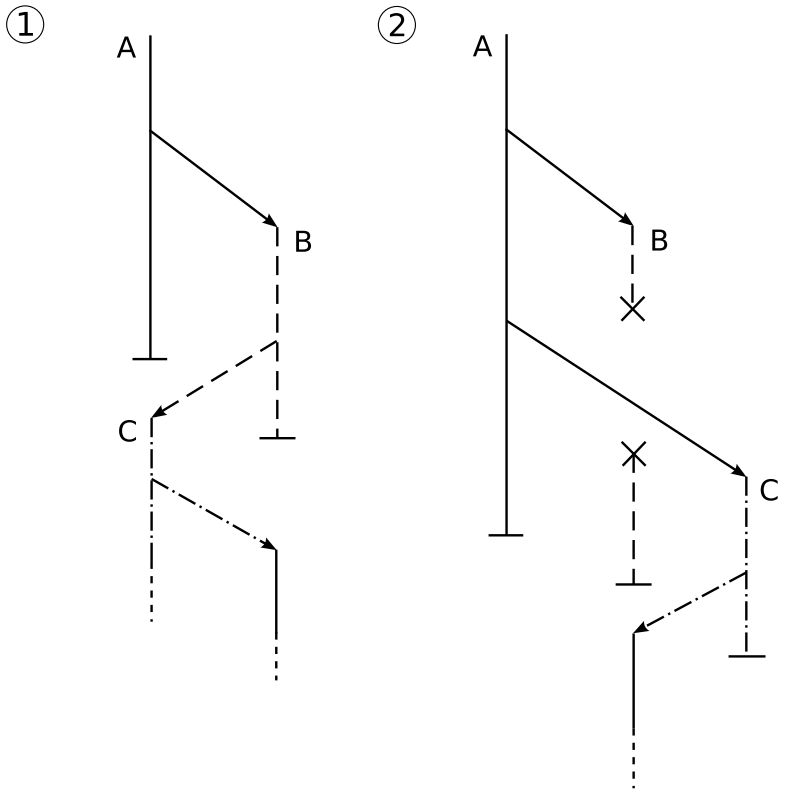

Figure 3.1: Scheduling actors on worker threads.

3.4.1 Implementation

Unfortunately, it is impossible for user-level code to find out if a thread running on the JVM is blocked[2]. We therefore implemented a simple heuristic that tries to approximate if a worker thread which executes an actor is blocked, or not.

The basic idea is that actors provide the scheduler with signs of life during their execution. That is, on every `send` and `receive` they call a `tick` method of the scheduler. The scheduler then looks up the worker thread which is currently executing the corresponding actor, and updates its time stamp. When a new receiver task item is submitted to the scheduler, it first checks if all worker threads are blocked. A worker thread with a "recent" time stamp is assumed to be not blocked. Only if all worker threads are assumed to be blocked (because of old time stamps), a new worker thread is created. Otherwise, the receiver task item is simply put into a queue waiting to be consumed by a worker thread that finished executing its task item.

Note that using the described approximation, it is impossible to distinguish blocked threads from threads that perform a long-running computation. Thus, a worker thread executing a compute-bound actor is assumed to be blocked. Consequently, compute-bound actors occupy their worker thread until they, finally, execute a blocking operation. This means basically, that compute-bound actors execute on their own thread.

At some point, the scheduler might decide to abandon idle worker threads, thereby freeing resources. Alternatively, worker threads might be kept alive, thereby reducing thread creation costs. We implemented a simple strategy, where newly created worker threads are kept alive indefinitely. However, for some applications it might be worth using a scheduler which optimizes the number of spare worker threads depending on runtime profiles. User-defined schedulers are easy to implement and use with our library.

[2]Using the Java Debug Interface of the Java Platform Debugger Architecture [SMa] more detailed information about the status of a VM thread can be obtained. However, the scheduler would have to be implemented as a debugging client, inspecting an application running on an external JVM. Also, it is not clear what the overhead of communicating with the JVM is.

In summary, additional threads are created only when needed to support (unexpected) blocking operations. The only blocking operation that we can handle without thread support is `receive`. Thus, a large number of non-cooperative actors (those using blocking operations other than what our library provides), may lead to a significant increase in memory consumption as the scheduler creates more and more threads. On the other hand, our approach adds significant flexibility, as the library does not need to be changed when the user decides to use a new blocking operation.

3.5 Timeouts

In this section we want to describe the handling of timeouts. Following our philosophy of a very light-weight approach, we do not want to spend a separate VM thread for running a timer process that keeps track of when and which actor to wakeup and notify of any timeouts. Moreover, by integrating timeout handling into our *rescheduling send*, the implementation of `receive` does not need to be changed.

In short, `receiveWithin` remembers the time when it was called and the specified duration after which the actor should execute the action associated with the `TIMEOUT()` pattern. These values are irrelevant when there is a matching message in the mailbox that can be processed immediately (as the actor immediately leaves the `receiveWithin`). When there is no matching message two cases need to be considered: (1) If the specified timeout is zero milliseconds (or negative) the timeout action should be executed immediately (receiving process is not suspended), and (2) if the specified timeout is positive, together with the continuation function, we remember that we need to check for timeouts.

Actual handling of timeouts occurs in the implementation of `send` as this is the first (and only) point where a receiving actor is activated after possibly being suspended for some time (rescheduling nature of send). Executing a specified timeout action is handled by transforming the message-to-be-received into an instance of the `TIMEOUT` case class. Otherwise, the code for scheduling a receiver task remains the same. Alternatively, timeout actions could be executed via *direct dispatch*, i.e. without scheduling a receiver task. This behavior could be justified by arguing that timeout actions typically execute short back-off code.

3.6 Event-Driven Applications

In our discussion of event-based actors so far, we were only concerned with point-to-point communication. For some applications a programming model based solely on asynchronous message passing is too low-level. A model where components (1) can express their interest in being notified when a certain type of event occurs (*subscribe*), and (2) are able to trigger events (*publish*) has been proven to help structuring state transitions in interactive programs. For example, virtually all libraries for programming graphical user interfaces (GUIs) expect applications to conform to some form of this model. However, standard approaches often cause the interactive logic of a program to be fragmented across multiple handlers or classes ("inversion of control"). Moreover, control flow among fragments is expressed only indirectly through modifications to state shared between handlers.

In the following we show how event-based actors can help in implementing a powerful abstraction for publish/subscribe-style programming. Recent work by Chin and Millstein [CM06] discusses "responders," a control-flow abstraction for *event-loops* that avoids many of the drawbacks of inversion of control and the state design pattern [GHJV95]. We introduce a similar *event-loop* abstraction built on top of event-based actors which removes some of the limitations of their approach.

An event-loop dispatches on a signaled event to handle it appropriately. Ordinary control-flow constructs may be used to transfer control between (possibly nested) event-loops. An event can be any message. A message is treated as an event if the receiving actor is blocked executing an event-loop. In contrast to "normal" messages, events are dropped (i.e. immediately removed from the mailbox) if there is no appropriate handler in the enclosing event-loop.

Figure 3.2 shows our solution to the "guessing game" using event-loops. Each of the event-loops represents a state of an instance of the game. The outer event-loop of the `start` method represents the state of a game that has not been started, yet. Thus, every signaled event which is different from `StartGame()` has no effect on the state, but causes the game to publish a `HaveNotStarted()` event. Instead, an event of type `StartGame()` will transfer control to a nested event-loop which represents the state of a newly started game that is ready to respond to the player's guesses. The

```
class GuessingGame extends Publisher {
  case class StartGame()
  case class Guess(num: int)
  val rand = new Random()
  def start =
    eventloop {
      case StartGame() =>
        val correctAnswer = rand.nextInt(50)
        eventloop {
          case Guess(guess) =>
            if (guess > correctAnswer)
              publish(GuessLower())
            else if (guess < correctAnswer)
              publish(GuessHigher())
            else {
              publish(GuessRight())
              start
            }
          case _ => publish(HaveNotFinished())
        }
      case _ => publish(HaveNotStarted())
    }
}
```

Figure 3.2: An implementation of the guessing game using our actor-based event-loop.

inner event-loop is not left until the player guesses the correct answer, at which point control is transferred to the outer event-loop (by a recursive call of `start`), thereby resetting the state.

Note that immediately after calling `start`, the game actor is suspended, causing the call-stack to be unwound. Thus, even a long-running session with many games started and finished will not cause the call-stack to overflow.

Having introduced our event-loop abstraction by a series of examples, we now want to more formally define its semantics by giving a definition in terms of the `receive` communication abstraction. An event-loop is a function `eventloop` of type `(PartialFunction[Message, unit] => scala.All)`. It takes a set of event handlers as its argument (note that a *single* event handler has the same type). As `eventloop` contains a (last) call to `receive`, it shares the property of not returning normally, thus having return type `scala.All` (see above). An event-loop

```
eventloop {
  case P1 => H1
  ...
  case Pn => Hn
}
```

is equivalent to

```
def _f = {
  case P1 => H1; receive(_f)
  ...
  case Pn => Hn; receive(_f)
  case _ => receive(_f)
}
receive(_f)
```

where $P_1 \ldots P_n$ are Scala patterns and $H_1 \ldots H_n$ are closures (of type `(=> unit)`). The default case which is listed last in the definition of `_f` makes sure that events which are not handled are immediately removed from the actor's mailbox. The given definition is surprisingly simple. There are several reasons: First, the pattern matching mechanism of `receive` (which, in turn, is provided by Scala's partial functions) can be

```
class Handler(a: Actor,
              f: PartialFunction[Message,unit])
  extends PartialFunction[Message,unit] {
  def isDefinedAt(m: Message): boolean = true
  def apply(m: Message): unit = {
    if (f.isDefinedAt(m)) f(m)
    else {
      // do nothing
    }
    a receive this
  }
}
```

Figure 3.3: Definition of a proxy handler for function arguments passed to `eventloop`.

readily reused. By introducing a catch-all pattern as the last case in `_f` we turned the actor's mailbox into an event-queue which is managed by `receive`. Also, events arriving during the execution of a particular handler are automatically queued, including events signaled on the executing actor itself. In contrast, Chin and Millstein's event-loop throws a runtime exception if this happens.

The definition of event-loops in terms of `receive` given above is not suitable for direct implementation as it would require a preprocessor or modifications to the Scala compiler.

Fortunately, the needed partial function can be created dynamically. Subclasses of `PartialFunction[-A,+B]` may override the `isDefinedAt()` and `apply()` methods, thereby defining a custom class of partial functions. Individual partial functions are obtained through instantiation.

Figure 3.3 shows the definition of a class of partial functions of the form of `_f` (in the following referred to as "handler"). The `Handler` class basically defines a *proxy* [GHJV95] for functions that are passed as arguments to `eventloop`.

Note that the receiving actor has to be explicitly passed as a constructor argument, because **`this`** inside the `apply()` method refers to the current handler instance. The definition of `eventloop` then merely consists of a

call to `receive` passing a new instance of our custom handler:

```
def eventloop(f: PartialFunction[Message,unit]): scala.All =
  receive(new Handler(this, f))
```

Chapter 4

Runtime System

Our programming model should help programmers build systems at a higher level of abstraction. Unnecessary architectural details, such as network connection management, should be hidden from the programmer. We aim to achieve this by providing high-level operations, such as spawning an actor on a remote node, which form part of a *virtual architecture*. The disparities between the virtual architecture and the underlying network of computers are resolved by a *runtime system*. The runtime system provides

- Location-transparent message delivery,
- Name registering and translation,
- Remote actor creation, and
- Location-transparent error handling

as services for the distributed execution of actors.

4.1 Architecture

The architecture of the runtime system consists of two layers. The layer on top is independent of the underlying network protocol (analogous to

the architecture described in the Erlang book we call it "NetKernel"). The layer at the bottom ("service" layer) is specific to the network protocol used. Factoring out architecture-independent parts makes our runtime system *portable*.

4.1.1 Service Layer

The service layer provides

- Remote message delivery,
- Serialization,
- Creation of process identifiers, and
- Connection management

as protocol-specific services.

How a message is delivered to a remote actor executing on a different node in the network ultimately depends on the underlying network protocol. However, before we can explain the details of remote message delivery, it is necessary to introduce process identifiers and name resolution.

Our message passing abstractions are built on operations that permit the transmission of sequences of bytes to processes running on remote computers. Messages are represented as normal object structures on the heap. Thus, to pass a message to a remote process, the object structure has to be turned into a sequence of bytes in such a way that it is possible to reconstruct an equivalent object structure at the other end. This kind of object encoding is exactly what serialization is concerned with.

4.1.2 Process Identifiers

Process identifiers (PIDs) are globally unique names for individual actors. They contain a (globally unique) node name which identifies the node on

which the actor is running, as well as an identifier used to look up a reference to the actor inside the (local) node.

As the constituents of the node name depend on the underlying network protocol (e.g. a node name suitable for TCP contains a port number, which is useless for a name identifying a JXTA node), we decided to create PIDs through a *factory method* [GHJV95] in the service layer.

The service layer also handles connection management. For some network protocols, such as the JXTA peer-to-peer protocol, the notion of connection has no meaning. As we largely want to support the programming model of distributed Erlang, we nevertheless require service layers to implement a set of connection related functions. Basically, this set consists of Erlang's built-in-functions `connect`, `disconnect`, `isConnected` and `nodes`.

For protocols with automatic discovery mechanisms the first two functions would simply return without doing anything. The result of `isConnected` would depend on whether the remote node can be discovered successfully. The set of nodes returned by `nodes` might include all nodes that have been discovered in the past.

4.2 Distributed Name Table

The user may register names for PIDs of actors. This additional layer of indirection offers a way to more abstractly address services running on a remote node without knowing the PID of a special actor. An actor can then be referred to using its registered name and the node on which it is residing.

For example,

```
register('rpc, self)
```

registers the actor which is evaluating this expression (self) on the local node as "rpc". A remote node can send messages to this actor in the following way:

```
name(node, 'rpc) ! msg
```

where node is the node local to the "rpc" actor.

Note that names are registered only locally to some node. Thus, the (virtual) name table used for translating (local) names to actors is distributed among all nodes. Global name services may be implemented on top of the distributed name table, of course (e.g., see [AVWW96]). We do not provide an implementation, however.

4.3 Message Delivery Algorithm

Location-transparent message delivery, just as any other service provided by the runtime system, is only available when accessing an actor through its PID. For true location-transparency it is necessary that all actors are accessed through their PIDs, even if they are local to each other. This way, program development is simplified, as the programmer does not have to remember if a specific actor is local or not.

As the general message delivery algorithm involves many indirections which might result in an unacceptable slowdown for local message sends, we devise an optimization for this special case (see 4.3.1).

To send a message to an actor, the sender calls the "!" method on the PID of the receiver $rpid$ passing the message to be sent msg as an argument (sends using registered names are similar). This results in a call to the NetKernel passing $rpid$ as well as msg.

If $rpid$ denotes an actor residing on the node of the NetKernel, a reference to the actor a is looked up in a local table, and the message is sent by calling `send` on a. In summary, using this algorithm, a local message send takes three method calls, one comparison and a table look-up.

If $rpid$ denotes an actor running on a remote node, a method of the underlying service is invoked passing $rpid$ and msg. The service first serializes msg into $msgSer$. Then, it creates a message `Send(rpid, msgSer)` which, in turn, gets serialized into $sendMsg$. A low-level `send` method of type `Node => (String => unit)` then delivers $sendMsg$ as a string to the remote node. The remote node first needs to deserialize the incoming byte stream and interpret the `Send` message. The rest proceeds as in a local

send.

We want to point out a crucial issue when serializing PIDs. A PID sometimes needs to access the services provided by a NetKernel (e.g. for remote message sends). Thus, PIDs hold a reference to the underlying NetKernel. This reference has to be updated to point to the local NetKernel whenever the corresponding PID gets deserialized.

4.3.1 Fast Local Sends

Updating references at deserialization time is a generally useful technique and is also the basis for an optimization we implemented for local message sends. The idea is that a PID additionally contains a *direct reference* to its actor. This reference can always be used when the PID and its actor reside on the same node. Of course, the reference has to be set to `null` whenever the PID gets deserialized on a node which is not local to the PID's actor. On the other hand, if we detect that a PID is deserialized on the node of its actor we look up the reference to the actor and set it accordingly. With this optimization in place, a local send involves only a null pointer check and a call of the `send` method of the actor.

4.4 Error Handling

In this section we want to discuss abstractions for error handling which our programming model supports. First, actors can be *linked* together, such that actors receive messages when one of their linked actors behaves in a faulty way. This way, other actors can be monitored and appropriate actions taken when they terminate abnormally.

Second, we devise a variant of asynchronous exceptions for event-based actors. Asynchronous exceptions allow familiar error handling methods to be used in a concurrent context.

4.4.1 Linked Actors

Our runtime system supports a mechanism for implicit communication between *linked actors*. The basic idea is that actors terminate or get appropriate error messages when one of their linked actors terminates.

Links are used in Erlang to enable the construction of fault-tolerant systems. By fault-tolerance we mean the ability to react to events caused by faulty behavior of (concurrent) components of a system. For this, the runtime system needs (1) to be able to detect faulty behavior, and (2) to signal special events to components interested in handling them. Erlang supports these mechanisms in a location-transparent way.

Our actor library provides an error handling mechanism equivalent to that of Erlang. In the following we informally quote the semantics of Erlang-style process links (substituting "actor" for "process" to be consistent with the rest of this text). After that, we show how links are implemented inside our runtime system.

Link Semantics

During execution, actors can establish links to other actors. If an actor terminates (normally or abnormally)[1], an *exit signal* is sent to all actors which are currently linked to the terminating actor. An exit signal has type

```
case class Exit(from: Pid, reason: Symbol) extends SystemMessage
```

The PID `from` denotes the terminating actor, `reason` is any Scala `Symbol` (basically a string).

The default behavior of an actor receiving an exit signal where `reason` is not `'normal` is to terminate and send exit signals to all actors to which it is currently linked. Exit signals where `reason` equals `'normal` are ignored by

[1] An actor terminates *normally* when (1) it either finishes execution, or (2) it evaluates `exit('normal)`. It terminates *abnormally* when (1) it does not handle a thrown exception, or (2) it calls `exit(reason)` where `reason` is not `'normal`.

default. An actor can override its default handling of exit signals, thereby receiving exit signals as normal messages.

Note that we distinguish exit *signals* from exit *messages*. Exit signals are messages which have a special meaning and are treated in a special way by the runtime system. An actor receives exit messages (which are not treated specially) only if it registers its interest to do so with the runtime system.

Establishing Links

An actor can establish a bidirectional link to another actor with PID `p` by evaluating `link(p)`. If the two actors are already linked together, this evaluation has no effect. If `p` is invalid, i.e. referring to a nonexistent actor, the runtime system sends a message `Exit(from, 'invalidPid)` to the actor evaluating `link(p)`.

Evaluating `spawnLink(block)` creates a new actor which is automatically linked to the evaluating actor. The behavior of the new actor is specified by `block`, a closure of type `RemoteActor => unit`.

For example,

```
spawnLink(a => {
  a.receive {
    case M(p) => p ! Ack()
  }
})
```

creates an (anonymous) actor which acknowledges the reception of a simple message. Note that, because Scala is *statically scoped*, inside the body of the passed closure, `this` is bound to the actor evaluating `spawnLink`. To reference the newly created actor, the closure argument `a` has to be used. Evaluating `spawnLink` behaves as if it had been defined as:

```
spawnLink(block: RemoteActor => unit) = {
  val p = spawn(block)
  link(p)
  p
}
```

However, `spawn` and `link` have to be executed atomically. If this was not the case, the newly spawned actor (defined by `block`) could be killed (e.g. by an exit signal), before the spawning actor evaluates `link`. Thus, the spawning actor could receive two *different* exit messages depending on whether the spawned actor was killed between `spawn` and `link` (reason "invalid PID") or after the execution of `spawnLink`.

Removing Links

By evaluating `unlink(p)` an actor `q` removes its link to `p` (if any). The link is removed bidirectionally, i.e. `p` also removes its link to `q`.

To ensure that exit messages are not sent to nonexistent actors, all links are removed if an actor terminates.

Runtime Support for Links

Establishing a link between a source actor and a target actor proceeds as follows: First, the local NetKernel registers an unidirectional link between the (local) source and the target (note that an actor evaluating `link(p)` for some PID `p` is necessarily local to the underlying NetKernel). Basically it maintains a hash table which maps local actors to sets of, possibly remote, PIDs. If the target actor is local, the NetKernel can directly create a bidirectional link (which is represented by two unidirectional links). Otherwise, it sends a system message to the remote NetKernel, asking for an additional unidirectional link. The unlinking procedure is analogous.

More interesting is the handling of terminating actors in the face of actor links. Whenever it is possible for the runtime system to detect if an actor terminates, it acts as if the actor had called `exit(reason)` for some symbol `reason`. For example, the `start` and `receive` methods catch all exceptions, so they can call `exit` supplying some reason depending on the exception as an argument.

Every call to `exit(reason)` is passed down to the NetKernel. It basically traverses the graph of linked actors in depth-first order, thereby coloring nodes, so as to detect cycles. For each visited PID it proceeds as follows:

For local actors (i.e. whose node is equal to the node of the NetKernel), we first check if it has overridden the default behavior. If it is marked as "trapping exits", the exit signal is converted to a normal message which is simply send to the actor. Otherwise, if `reason` equals `'normal` the exit signal is ignored (by default, exit signals with reason `'normal` only cause the original actor to terminate). For all other values of `reason`, runtime information about the actor is removed from internal tables of the NetKernel, and `exit` is called on all transitively linked actors.

If the visited PID denotes a non-local actor, a special system message is sent to its remote NetKernel. As all (bidirectional) links are removed on termination, this message also contains the source of the link.

4.4.2 Asynchronous Exceptions

The basic idea of asynchronous exceptions is to be able to handle exceptions that are thrown inside a concurrent process. That is, for two different processes A and B, A shall be able to handle an exception which is thrown in B but not handled by B. Programming models which lack such a mechanism can be hard to use. In the following we devise an exception model suitable for the case when concurrent processes are actors.

Once actors are running, interaction between them is basically limited to sending messages (`receive` is a local operation on the private mailbox), and establishing and removing links (except for local actors which, additionally, are normal objects). The exception model we describe in the following is consistent with this model of interaction. It merely adds the ability to install exception handlers when spawning new actors.

For example, a supervising actor might want to restart its slave when it terminates because of an unhandled exception:

```
def startSlave() =
  trySpawn { a =>
    a.doWork()
  }
  acatch {
    case t: Throwable => startSlave()
  }
```

`trySpawn` is a method of class `Actor` which can be used like a new kind of statement[2]. Note that `trySpawn` is non-blocking. Therefore, the exception handler "goes out of scope" before an exception is thrown.

The semantics of trySpawn

When an exception is thrown, the nearest dynamically enclosing exception handler is located. Exception handlers are arguments of `acatch`. Because `trySpawn` may be used to spawn new actors on remote machines, an exception may be thrown on a different machine than where the corresponding exception handler is installed. Thus, exception handlers are referred to using exception handler descriptors.

Each actor maintains a stack of exception handler descriptors. Initially, this stack is empty. Actors which are spawned by a "parent" actor, inherit their parent's top of the stack (if any). Evaluating `trySpawn` creates a new descriptor which is associated with the specified handler and pushed onto the stack. Each handler contains (a copy of) their parent's descriptor (if any). Consequently, the nearest dynamically enclosing exception handler descriptor is always the top of the stack.

Handling an exception, using the data structures described above, then proceeds as follows: If the stack of exception handler descriptors is empty, the actor terminates abnormally by calling `exit` (providing a string representation of the exception as an argument). This way, exceptions are integrated with actor links. Otherwise, we obtain the descriptor of the nearest dynamically enclosing exception handler which is the top of the stack. If the corresponding handler is local, it is simply executed. Otherwise, the NetKernel sends a special message to the remote node where the handler is installed.

[2]We achieve this by combining two basic language properties of Scala: First, any method may be used as an infix or postfix operator. Thus, `trySpawn {...} acatch {...}` is short for `trySpawn({...}).acatch({...})`. That is, `trySpawn({...})` creates an object with a method `acatch`. Second, closures are constructed automatically depending on the expected type. In the example, `{ a => a.doWork() }` is automatically converted into a closure of type `(RemoteActor => unit)`.

4.5 Type-Safe Serialization

Messages are usually represented as object structures on the heap. Before they can be sent over the network, their object graph needs to be *serialized* (or *pickled*, or *marshalled*) into a stream of bytes. Many programming languages and runtime environments provide general serialization and deserialization mechanisms. Usually, these serialization mechanisms rely on *introspection* (or *reflection*), i.e. the ability to inspect the contents and structure of object types at run-time. For instance, the Java programming language provides a standard API for object serialization which depends on reflection. Not every Java virtual machine (JVM) supports reflection: The KVM [SMb], a Java virtual machine suited for resource-constrained devices, does not include a reflection framework because the additional space requirements would be prohibitive (the KVM is suitable for devices with a total memory budget of no more than a few hundred kilobytes).

As the implementation of our distributed programming model shall be suited for mobile and resource-constrained devices possibly running restricted JVMs, such as the KVM, we want to avoid a general reflective mechanism that crucially depends on the underlying virtual machine. Instead, we provide a combinator library for constructing *picklers* for user-defined data types. A pickler is a pair consisting of a pickling and an unpickling function.

As outlined by Elsman [Els04], pickler combinators have the following desirable properties:

1. Compactness of pickled data due to type-specialization. The type of the pickler contains all the information necessary to know how to pickle and unpickle values of a special type. Thus, no type information has to be emitted during pickling.

2. Compactness of pickled data due to structure sharing. Existing picklers can be wrapped in a meta combinator which pickles multiple copies of the same object structure only once. Unpickling of values pickled using structure sharing leads to heap compaction.

3. Type safety. The pickling function of a pickler for type t can only be applied to values of type t. Likewise, the unpickling function of a pickler for type t can only return values of type t.

4. No need for runtime type tags. Since the library is written in Scala, it is possible to define picklers specialized for every possible Scala type. No runtime type tags are necessary, as picklers are statically defined.

The user has to provide picklers for every type t, such that there may exist a value of type t which may be sent to a remote node. With our combinator library we strive to make the construction of user-defined picklers as simple as possible.

In the following, we first want to present the library interface. By means of small examples we show how to use combinators provided in the library. Then, we describe our implementation in Scala, focusing on the meta combinator for structure sharing. Finally, we describe how the combinator library is used inside the runtime system.

4.5.1 Library Interface

The combinators provided in the library construct picklers of type `PU[t]`. A pickler of type `PU[t]` encapsulates both pickling and unpickling functions in a single value. These functions are not directly used by an application. Instead, two polymorphic functions `pickle` and `unpickle` provide a high-level interface to them. They have the following signatures:

```
def pickle[t](p: PU[t], a: t): String
def unpickle[t](p: PU[t], stream: String): t
```

To keep the presented code simple, for now our picklers read and write strings. In our runtime system, streams are used, instead (see 4.5.3).

Figure 4.1 shows the function signatures of our library combinators. There are basically two groups of combinators: (1) primitive combinators (for types such as `int` and `string`), and (2) pickler *constructors* for tuple types (`pair`, `triple` and `quad`), lists (`list`), etc.

Finally, there exist two special pickler constructors which not only consume picklers, but also functions: `wrap` consumes two functions for pre- and post-processing, respectively. `data` consumes a tagging function used

```
def unit: PU[unit]
def int: PU[int]
def bool: PU[boolean]
def char: PU[char]
def string: PU[String]

def pair[a, b](PU[a], PU[b]): PU[Pair[a, b]]
def triple[a, b, c](PU[a], PU[b], PU[c]): PU[Triple[a, b, c]]
def quad[a, b, c, d](PU[a], PU[b], PU[c], PU[d]): PU[Tuple4[a, b, c, d]]
def list[a](PU[a]): PU[List[a]]
def option[a](PU[a]): PU[Option[a]]
def wrap[a,b](a => b, b => a, PU[a]): PU[b]
def data[a](a => int, List[() => PU[a]]): PU[a]
```

Figure 4.1: The pickler interface.

to identify disjoint subsets of a type. The former combinator is suited for pickling record-style types (without variants). The latter handles recursive algebraic data types.

For example, a pickler for integer-pair lists is constructed by combining the `int`, `pair` and `list` combinators:

```
val puPairs = list(pair(int, int))
```

Suppose we want to pickle values of the following Scala type:

```
case class Person(name: String, age: int)
```

Pickling pairs of strings and ints is easy (by using the combinator `pair(string, int)`). But, how do we make sure our pickler for `Person` refuses to pickle a `Cat("Billy", 8)` which can be represented similarly? For this, we use the `wrap` combinator. It takes two conversion functions which make sure values are deconstructed and reconstructed in a type-safe way:

```
val personPU = wrap(p: Pair[String,int] => Person(p._1, p._2),
                    p: Person => Pair(p.name, p.age),
                    pair(string, int))
```

As personPU has type PU[Person] it is impossible to apply it to values of the wrong type.

Picklers for algebraic data types are constructed using the data combinator. Only data types that are not mutually recursive with other data types are supported. For any number of mutually recursive data types it is possible to define a suitable combinator, though (see [Els04] for example). Given an algebraic data type t with n case class constructors C_1, ..., C_n, a pickler of type PU[t] may be constructed using the data combinator, passing (1) a function mapping a value constructed using C_i to the integer i ($0 <= i < n$) and (2) a list of functions $f_0, \ldots, f_{n-1}$, where each function f_i has type () => PU[C_i].

So, for instance, consider a pickler for terms in the untyped lambda calculus:

```
abstract class Term
case class Var(s: String) extends Term
case class Lam(s: String, t: Term) extends Term
case class App(t1: Term, t2: Term) extends Term

def varPU: PU[Term] =
  wrap(Var,
       t: Term => t match { case Var(x) => x },
       string)

def lamPU: PU[Term] =
  wrap(p: Pair[String, Term] => Lam(p._1, p._2),
       t: Term => t match { case Lam(s, t) => Pair(s, t) },
       pair(string, termPU))

def appPU: PU[Term] =
  wrap(p: Pair[Term, Term] => App(p._1, p._2),
       t: Term => t match { case App(t1, t2) => Pair(t1, t2) },
       pair(termPU, termPU))

def termPU: PU[Term] = data(t: Term => t match
  { case Var(_) => 0; case Lam(_,_) => 1; case App(_,_) => 2 },
  List(() => varPU, () => lamPU, () => appPU))
```

4.5.2 Implementation

As mentioned before, picklers aggregate a pair of functions, for pickling and unpickling, respectively, in a single value. A pickler of type `PU[t]` can pickle and unpickle values of type `t`. The abstract type `PU[t]` is given by the following definitions:

```
abstract class PU[t] {
  def appP(a: t, state: PicklerState): PicklerState
  def appU(state: UnPicklerState): Pair[t, UnPicklerState]
}
abstract class PicklerState(val stream: OutStream,
                            val dict: PicklerEnv)
abstract class UnPicklerState(val stream: InStream,
                              val dict: UnPicklerEnv)
abstract class PicklerEnv extends HashMap[Any, int]
abstract class UnPicklerEnv extends HashMap[int, Any]
```

The pickling function `appP` consumes a value of type `t`, thereby transforming the state of the pickler. Conversely, the unpickling function `appU` produces a value of type `t`, and transforms the unpickler state in the process. Besides a stream of bytes produced so far, the state manipulated during pickling also includes a *pickler environment*. The pickler environment is used for structure sharing (see 4.5.2). Similarly, the unpickling function needs to maintain an unpickler environment.

Now, to the picklers themselves. Picklers for base types such as integer simply have to implement the pickling and unpickling functions, thereby producing or consuming bytes. We leave the handling of environments to a special `share` meta combinator.

As an example, consider a pickler for characters:

```
def char: PU[char] = new PU[char] {
  def appP(b: char, s: PicklerState): PicklerState = {
    s.stream.write(b)
    s
  }
  def appU(s: UnPicklerState): Pair[char, UnPicklerState] =
    Pair(s.stream.readChar, s);
}
```

Picklers for integers and booleans are implemented similarly. Picklers for strings are implemented as picklers for character lists. However, before we can explain the pickling of lists, we need to introduce two basic combinators.

The first, `lift`, produces a pickler which can only pickle and unpickle a single value, leaving all state unchanged:

```
def lift[t](x: t): PU[t] = new PU[t] {
  def appP(a: t, state: PicklerState): PicklerState = state;
  def appU(state: UnPicklerState) = Pair(x, state);
}
```

At first sight, this pickler might seem to be useless. We will see, however, that it is needed in the definition of picklers for tuple types.

The second basic combinator, `sequ`, encapsulates sequential pickling and unpickling:

```
def sequ[t, u](f: u => t, pa: PU[t], k: t => PU[u]) = new PU[u] {
  def appP(b: u, s: PicklerState): PicklerState = {
    val a = f(b)
    val sPrime = pa.appP(a, s)
    val pb = k(a)
    pb.appP(b, sPrime)
  }
  def appU(s: UnPicklerState): Pair[u, UnPicklerState] = {
    val resPa = pa.appU(s)
    val a = resPa._1
    val sPrime = resPa._2
    val pb = k(a)
    pb.appU(sPrime)
  }
}
```

The basic idea is, that a value `b` of type `u` is pickled by sequentially applying two picklers `pa` and `pb`, where `pb` depends on `b`. First, a *projection function* `f` is applied to the value to be pickled. The resulting value is pickled using the first pickler that is supplied as an argument, i.e. `pa`. Using the projected value, a pickler `pb` is obtained which is able to pickle the *original value*. Thus, the projected value is "only" pickled to provide enough in-

formation in the byte stream so that the correct combinator for unpickling the original value can be selected.

To demonstrate the `sequ` combinator, consider the definition of a combinator for pairs:

```
def pair[a, b](pa: PU[a], pb: PU[b]): PU[Pair[a, b]] =
  sequ(fst, pa, x => sequ(snd, pb, y => lift(Pair(x, y))));
```

The `fst` and `snd` functions are *projections* which map a pair to its first or second component, respectively. So, to pickle a pair `p` of type `Pair[a,b]`, we first apply `fst` to `p`, thereby projecting out the part that we can pickle using `pa`, i.e. the first component of `p`, namely `x`. Knowing `x`, the *entire* pair `p` can be pickled using the following pickler:

```
sequ(snd, pb, y => lift(Pair(x, y)))
```

How does this last pickler pickle the *entire* pair `p`? It only pickles `y`, the second component of `p`, because it already *knows* `x`. Thus, after having pickled `y`, nothing remains to be done. Therefore we use `lift` which results in a no-op. Conversely, when unpickling, `lift(Pair(x, y))` simply returns `Pair(x, y)`. It does not have to read from the stream, as the previous picklers already determined `x` and `y`.

The `wrap` combinator pickles values of type `b` by pre- and post-processing them using functions of type `b => a` and `a => b`, and using a pickler for `a` to do the actual pickling and unpickling:

```
def wrap[a, b](i: a => b, j: b => a, pa: PU[a]): PU[b] =
  sequ(j, pa, x: a => lift(i(x)));
```

Now we are ready to introduce picklers for lists. Lists are pickled by using the sequential combinator `sequ` to first pickle the list length (using `nat`, a pickler for natural numbers), and then a list of fixed length:

```
def list[a](pa: PU[a]): PU[List[a]] =
  sequ(l: List[a] => l.length, nat, fixedList(pa));
```

Note that `fixedList(pa)` is a function of type `int => PU[List[a]]`. Again, `sequ` expresses the fact that the special "`fixedList` pickler" used to pickle the list depends on the result of applying the projection function `(l: List[a])=>l.length` to the argument list.

Finally, `fixedList` is a family of picklers which pickle lists of known length as pairs:

```
def fixedList[a](pa: PU[a])(n: int): PU[List[a]] = {
  def pairToList(p: Pair[a,List[a]]): List[a] =
    p._1 :: p._2;
  def listToPair(l: List[a]): Pair[a,List[a]] =
    l match { case x :: xs => Pair(x, xs) }
  if (n == 0) lift(Nil)
  else
    wrap(pairToList, listToPair, pair(pa, fixedList(pa)(n-1)))
}
```

An important application of the list combinator is pickling of strings as character lists:

```
def string: PU[String] =
  wrap(List.toString, (str: String)=>str.toCharArray().toList, list(char)
```

Single Recursive Data Types

Compared to simple record data types, algebraic data types pose an additional problem: During pickling and unpickling the used case class constructors have to be recorded in the byte stream, so they can be applied in the correct order when reconstructing values. This is done using a tagging function tag which assigns distinct integers to the different alternatives. Using the sequential combinator `sequ`, values are pickled after their tags:

```
def data[a](tag: a => int, ps: List[() => PU[a]]): PU[a] =
  sequ(tag, int, x: int => ps.apply(x)());
```

Each function of type `()=>PU[s]` in the list we provide to the `data` combinator yields a pickler of type `PU[s]` when applied to the empty parameter list. This *lazy evaluation* is necessary because of the recursive construction of picklers. Consider our example of a pickler for terms in the untyped lambda calculus: If the picklers where eagerly constructed, `termPU` would construct a `lamPU` which would construct a `termPU` etc. ad infinitum.

Structure Sharing

In this section we describe a meta combinator which makes existing picklers aware of shared structures in the values to be pickled.

For example, when pickling the lambda term `kki`, the definition of `k` should be encoded only once:

```
val x = Var("x")
val i = Lam("x", x)
val k = Lam("x", Lam("y", x))
val kki = App(k, App(k, i))
```

The `share` combinator uses environment information to remember which values have been pickled at which location in the byte stream. As pickler and unpickler environments are threaded through all combinators, this information gets not destroyed by existing picklers. The `share` combinator takes any pickler as argument and produces a pickler of the same type:

```
def share[a](pa: PU[a]): PU[a]
```

The basic idea of the `share` combinator is the following: When pickling a value a, its location l in the byte stream is recorded in the pickler environment. Whenever a value equal to a is pickled, only a reference to the location of a is written to the output stream.

Figure 4.2 shows the implementation of the `share` meta combinator. Pickling a value `v` using structure sharing proceeds as follows: First, we check, if there is some value equal to `v` associated with a location `l` in the pickler environment. In this case, we write a "Ref" tag to the output stream together with `l`. Otherwise, a "Def" tag is written to the output stream. We record the current location of the output stream, pickle the value "normally", and add a mapping from `v` to `l` to the pickler environment.

Unpickling is simply the inverse operation: First, we read a tag. If we read a "Def" tag, a new value is about to be defined. Thus, we record the location `l` of the input stream. Then, the value is unpickled using the argument pickler. Finally, we add an entry to the unpickler environment mapping `l` to `v`. If the tag we read in the first step was a "Ref" tag, then we know that the following integer denotes a stream location. We use it to look up the resulting value in the unpickler environment.

```
def share[a](pa: PU[a]): PU[a] = new PU[a] {
  def appP(v: a, state: PicklerState): PicklerState = {
    val pe = state.dict
    pe.get(v) match {
      case None =>
        val sPrime = refDef.appP(Def(), state.stream)
        val l = pe.nextLoc()
        val sPrimePrime = pa.appP(v, new PicklerState(sPrime, pe))
        pe.update(v, l)
        return sPrimePrime
      case Some(l) =>
        val sPrime = refDef.appP(Ref(), state.stream)
        return new PicklerState(nat.appP(l, sPrime), pe)
    }
  }
  def appU(state: UnPicklerState): Pair[a, UnPicklerState] = {
    val upe = state.dict
    val res = refDef.appU(state.stream)
    res._1 match {
      case Def() =>
        val l = upe.nextLoc
        val res2 = pa.appU(new UnPicklerState(res._2, upe))
        upe.update(l, res2._1)
        return res2
      case Ref() =>
        val res2 = nat.appU(res._2)
        upe.get(res2._1) match {
          case None =>
            error("invalid unpickler environment")
            return null
          case Some(v) =>
            return Pair(v.asInstanceOf[a],
                        new UnPicklerState(res2._2, upe))
        }
    }
  }
}
```

Figure 4.2: A sharing combinator.

```
1 02           App(
1 01 1 01 x        Lam("x",
1 01 1 01 y            Lam("y",
1 00 0 43                  Var("x"))),
1 02               App(
0 42                   k,
1 01 0 43              Lam("x",
0 46                       x)))
```

Figure 4.3: Sharing example.

For example, we can apply the share combinator to our pickler for lambda terms:

```
def sharedTermPU: PU[Term] =
  share(data(t: Term => t match {
               case Var(_) => 0
               case Lam(_,_) => 1
               case App(_,_) => 2 },
             List(() => varPU, () => lamPU, () => appPU)))
```

Figure 4.3 presents an application of it to kki. The first and third columns (if applicable) show the tags used to track definitions and references, "1" denoting a definition and "0" denoting a reference. Note that the first definition (the top-level application term) is assigned stream location 41 hexadecimal (we allocate locations starting from 64, so they are easily recognized in the pickled data). The definition of the lambda term starting in the second line gets stream location 42 assigned. The variable x, on the same line, gets location 43 and so on. Note how both definitions of terms k and x are shared in the pickled data.

Using our structure sharing combinator, it is very easy to introduce a string table. In fact, the preceding example already made use of it (string "x" is defined at stream location 43 which appears twice in the pickled data). The original definition of our string pickler (see above) merely has to be surrounded by an application of share:

```
def string: PU[String] =
  share(wrap(List.toString,
             str: String => str.toCharArray().toList,
             list(char)))
```

4.5.3 Integration into the Runtime System

In this section we describe the integration of our pickler combinator library into the runtime system. Every message that needs to be sent to a remote node has to be serialized before it can be transmitted. The service layer provides a low-level function which handles transmission of strings to remote nodes. Thus, serialization of messages (object structures) into strings has to be done in the layer above, the NetKernel.

However, as we do not provide length information in the pickled data–it is included implicitly in the type of the pickler–deserialization has to be done right when the data gets read from the stream. The resulting message then gets sent to the NetKernel which unpacks it, possibly involving additional deserialization steps.

A serializer provides the following interface:

```
abstract class Serializer(kernel: NetKernel) {
  def serialize(o: AnyRef, w: Writer): unit
  def deserialize(r: Reader): AnyRef
  def pid: PU[Pid]
}
```

Serialization of a value `o` of type `AnyRef` proceeds by looking up a suitable pickler `repr` in an internal table which maps type names to picklers. To make sure the same pickler is selected at deserialization time, we start with encoding the type name (as all bytes are received from a stream, the length is encoded first). Then, we use the `appP` function of `repr` to write pickled data to the output stream.

Conversely, deserialization proceeds by first reading a type name from the input stream and looking up the pickler in an internal table. Actual unpickling is done by applying the `appU` function to the input stream. Even though the unpickled value has type `AnyRef`, no casts are necessary as the NetKernel reconstructs type information using pattern matching.

Picklers for PIDs are handled in a special way. Remember that PIDs need a reference to the (local) NetKernel in order to support location-transparent operations. This reference needs to be adjusted whenever a PID gets deserialized on a remote node. In the process, we also look up a direct reference

to the PID's actor if possible to support fast local sends:

```
def pid = new PU[Pid] {
  val nodeIntPU = wrap(p: Pair[TcpNode,int] =>
                         TcpPid(p._1, p._2, kernel,
                                if (p._1 == kernel.node)
                                  kernel.getLocalRef(p._2)
                                else null),
                       t: TcpPid => Pair(t.node, t.localId),
                       pair(tcpNodePU, int));
  def appP(pid: Pid, state: OutStream): OutStream =
    pid match {
      case tpid: TcpPid =>
        nodeIntPU.appP(tpid, state)
      case other =>
        error("TcpPid expected")
    };
  def appU(state: InStream): Pair[Pid,InStream] =
    nodeIntPU.appU(state);
}
```

Chapter 5

Evaluation

In this chapter we evaluate our event-based implementation of actors. In the first part, we carry out experiments to determine the performance of basic actor operations. In the second part, we evaluate our approach with respect to ease of programming. For this, we report on our experience in implementing a distributed auction service as a case study.

5.1 Performance

In this section we want to examine crucial performance properties of our event-based implementation of actors. In the process, we compare benchmark execution times of event-based actors with a state-of-the-art Java-based actor language, as well as with a thread-based version of our library. In some cases, we show the performance of straight-forward implementations using threads and synchronized data structures. Where possible we try to use benchmarks that were published before. In addition to execution time we are also interested in scalability with respect to the number of simultaneously active actors each system can handle. This type of scalability test will also tell us important facts about the memory consumption of actors in each system.

5.1.1 Experimental Setup

We present results from three different experimental settings:

Armstrong Test evaluates blocking operations in a queue-based application. We compare the execution times of three equivalent actor-based implementations, each written using a different actor language or library. In the process, we also examine how many actors each system can handle (for a given maximum heap size). We also show the performance of a naïve thread-based implementation.

Mergesort compares a thread-based implementation of our actor library (each actor executes on its own thread) with our event-driven implementation. We consider a concurrent implementation of the mergesort algorithm for sorting a linked list of long integers. To each sublist we assign an actor which is responsible for sorting it. For each list two more actors are created that (recursively) sort their respective sublists. The sorted sublists are sent back (as messages) to the respective parent actor which merges them.

Multicast Protocols compare the performance of basic actor operations in SALSA and event-based actors. We use three variations of multicast protocols which have been used to evaluate the performance of a previous version of SALSA, a state-of-the-art Java-based actor language [VA01]. However, we only compare with the latest version available at the time of this writing (SALSA 1.0.2). The used benchmarks are included in the SALSA distribution[1].

5.1.2 Armstrong Test

In the *Armstrong* test[2] we measure the throughput of blocking operations in a queue-based application. The application is structured as a ring of n producers/consumers with a shared queue between each of them. Initially, k of these queues contain tokens and the others are empty. Each

[1] Available at http://www.cs.rpi.edu/research/groups/wwc/salsa/index.html.

[2] named after Joe Armstrong, the creator of Erlang, who posed a similar benchmark as a challenge for concurrent programming languages at the Lightweight Languages Workshop 2002, MIT.

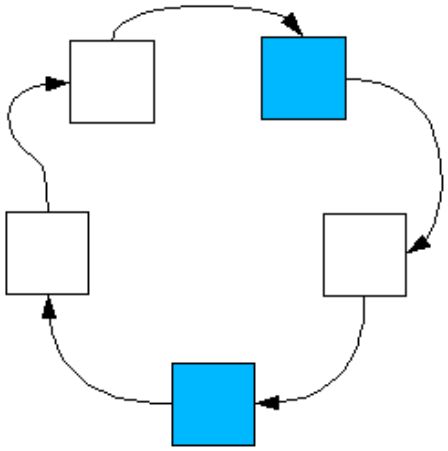

Figure 5.1: Armstrong test: Configuration of the queue-based application.

producer/consumer loops removing an item from the queue on its right and placing it in the queue on its left. Figure 5.1 illustrates this configuration.

We chose to include the Armstrong challenge in our performance evaluation for several reasons: First, all actors are created at once, allowing us to measure start-up time of the ring. That way, we can determine the actor creation time for each system. Second, it easy to test scalability–increase the ring size until the system crashes. Third, as blocking operations dominate, we expect to be able to measure overhead of context switches. We do not claim that the Armstrong test is a realistic application workload, though.

The following tests were run on a 1.60GHz Intel Pentium M processor with 512 MB memory, running Sun's Java HotSpot Client VM 1.5.0 under Linux 2.6.12. We set the JVM's maximum heap size to 256 MB to provide for sufficient physical memory to avoid any disk activity. In each case we took the median of 5 runs.

The execution times of three equivalent actor-based implementations written using (1) our event-based actor library, (2) a thread-based version of a similar library, and (3) SALSA [VA01], a state-of-the-art Java-based actor library, respectively, are compared.

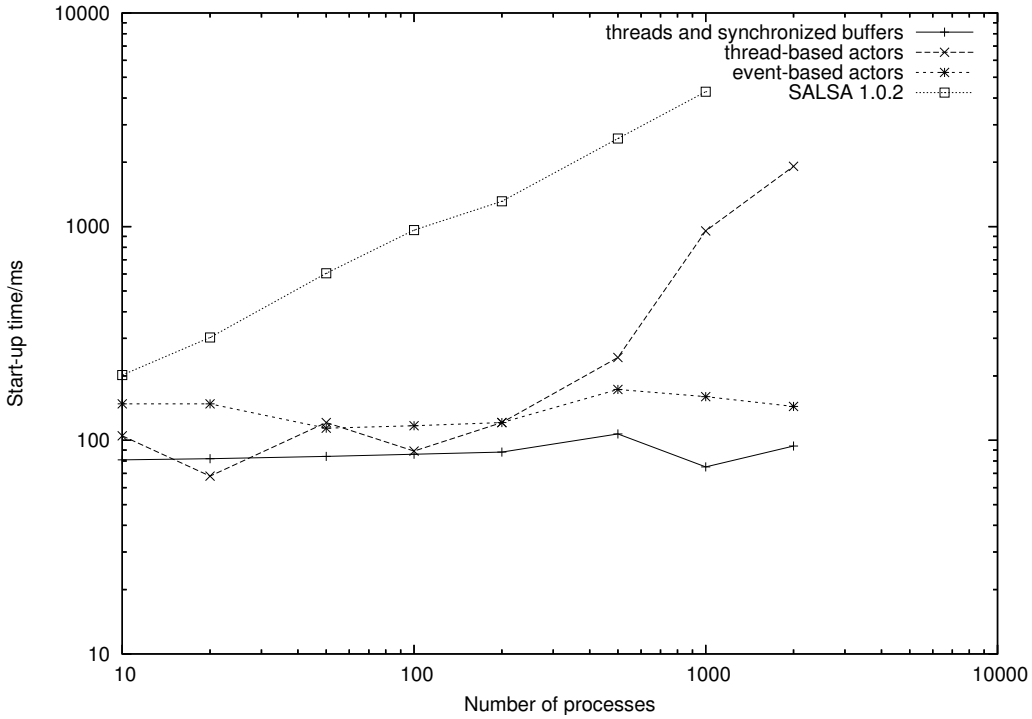

Figure 5.2: Start-up time.

Figure 5.2 shows start-up times of the ring for up to 2000 processes (note that both scales are logarithmic). For event-based actors and the naïve thread-based implementation, start-up time is basically constant. Event-based actors are about 60% slower than pure threads. For thread-based actors, start-up time increases exponentially when the number of processes approaches a thousand. With 4000 processes the JVM crashes because of exhaustion of maximum heap size. Using SALSA, the VM was unable to create 2000 processes. As each actor has a thread-based state object associated with it, the VM is unable to handle stack space requirements at this point. In contrast, using event-based actors the ring can be operated with up to 310000 processes that are created in about 10 seconds. Note that, because in the actor-based implementations every queue is an actor, the number of simultaneously active actors is actually two times the number of processes.

Looking at the generated Java code shows that SALSA spends a lot of time setting up actors for remote communication (creating locality descriptors, name table management, etc.), whereas in our case, an actor must announce explicitly that it wants to participate in remote communications (by calling `alive()`). Creation of locality descriptors and name table management can be delayed up to this point. Also, when an actor is created in SALSA, it sends itself a special "construct" message which takes additional time.

In summary, event-based actors provide an alternative to SALSA with inexpensive start-up times. Moreover, our event-based implementation can handle a number of actors two orders of magnitude higher than SALSA.

Figure 5.3 shows the number of token passes per second depending on the ring size. We chose a logarithmic scale for the number of processes to better depict effects which are confined to a high and strongly increasing number of processes. For up to 1000 processes, increase in throughput for event-based actors compared to pure threads averages 22%. As blocking operations clearly dominate, overhead of threads is likely to stem from context switches and contention for locks. Interestingly, overhead vanishes for a small number of processes (10 and 20 processes, respectively). This behavior suggests that contention is not an issue in this case, as uncontended lock management is optimized in Sun's HotSpot VM 1.5. Contention for locks becomes significant at about 2000 processes. Finally, when the number of processes reaches 4000, the threads' time is consumed managing the shared buffers rather than exchanging tokens through them.

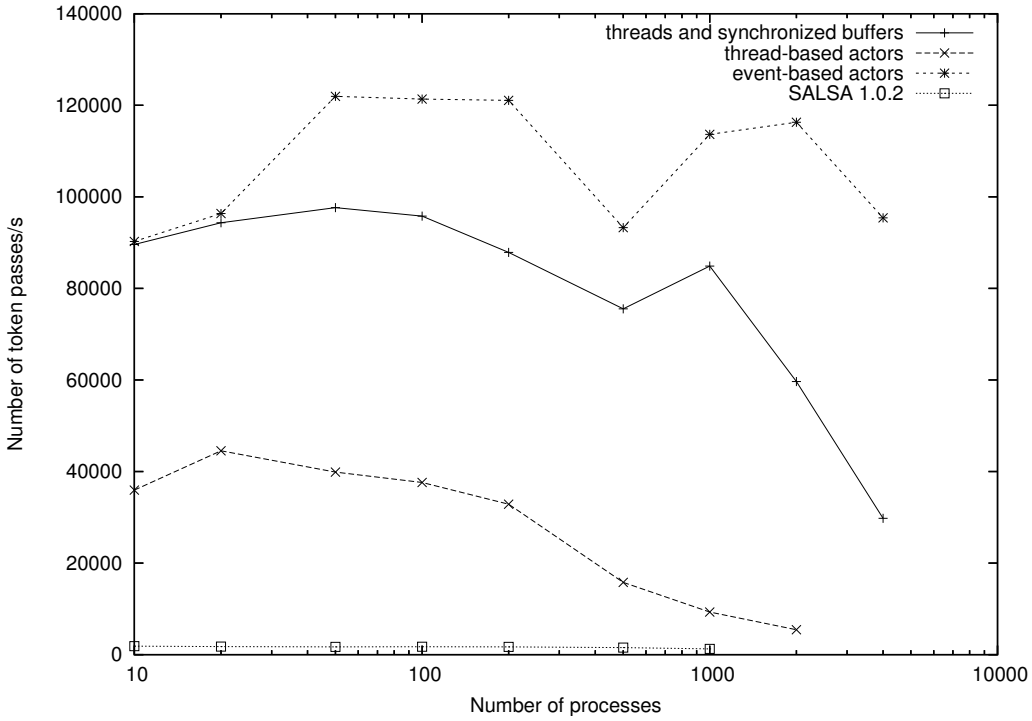

Figure 5.3: Measuring throughput (number of token passes per second) for a fixed number of 10 tokens varying the number of processes.

At this point throughput of event-based actors is about 3 times higher.

For SALSA, throughput is about two orders of magnitude lower compared to event-based actors. The average for 10 to 1000 processes amounts to only 1700 token passes per second. Looking at the generated Java source code revealed that in SALSA every message send involves a reflective method call. Using a simple benchmark published at IBM's "developerWorks" web site[3], we found reflective method calls to be about 30 times slower than JIT-compiled method calls on our testing machine.

For thread-based actors, throughput is almost constant for up to 200 processes (on average about 38000 token passes per second). At 500 processes it is already less than half of that (15772 token passes per second). Similar to pure threads, throughput breaks in for 2000 processes (only 5426 token passes per second). Again, contended locks and context switching overhead are likely to cause this behavior. The VM is unable to create 4000 processes, because it runs out of memory.

5.1.3 Mergesort

Figure 5.4 compares the performance of an actor-based mergesort implementation using our event-based library with a thread-based actor library. Note, that we spawn two more actors whenever the (sub-)list to be sorted has a length greater than one. For best performance results, a list with no more than, say, 256 elements would not be splitted but instead sorted with a sequential algorithm. We do not go this path because we mainly want to test how our implementation performs with a large number of actors.

All tests were run on an Intel Pentium 4 workstation (3 GHz hyper-threaded core), running Sun's Java HotSpot Client VM 1.5.0 under Linux 2.6.11. In each case we took the median of 3 test runs. In summary, our results with an event-driven implementation outperform an equivalent thread-based version.

For list sizes of 2048 or 4096 elements event-based actors running on 2 worker threads are about three times faster. As the stack space require-

[3]See article "Java programming dynamics, Part 2: Introducing reflection" at http://www-128.ibm.com/developerworks/library/j-dyn0603/.

len	thr	sgl	1wt	2wt	S(thr/2wt)
1024	1141	504	578	479	2.382
2048	3809	1197	1289	1193	3.193
4096	10792	3928	4132	3817	2.827
8192	(1)	14751	15494	14444	–

Figure 5.4: Performance of the mergesort test with varying list lengths. In each case we record the sorting time in ms. Column S(thr/2wt) lists the speedups between the thread-based version and the event-driven version with 2 worker threads.

ment at this point should not pose a problem to the operating system, we suppose that context switches between the actors' threads, which are especially frequent when lists are small, contribute to the difference in performance. When trying to run our thread-based mergesort benchmark with a list of 8192 elements the virtual machine crashed (1). As in this case the number of threads approaches several thousand (for a list of 8192 elements, theoretically there might be 16383 threads active simultaneously), the virtual machine is unable to create the requested number of native threads due to large stack space requirements. For a single thread the overhead of worker threads is less than 15% when sorting a list of 1024 elements. For larger lists the overhead is about 5%.

5.1.4 Multicast

In this section we use three variations of multicast protocols to compare the performance of basic actor operations in SALSA and event-based actors. For this, we ported the benchmarks included in the SALSA distribution to Scala using our actor library.

In the first multicast benchmark, a *director* sends a simple message to a number of actors. Messages are sent concurrently by spawning a separate actor for each message to be sent. The actors participating in the cast finish execution immediately after receiving their message.

Figure 5.5 shows the time (in milliseconds) needed for the multicast varying the number of receiving actors. For our actor library we include measurements for single-threaded actors as well as for scheduled actors (using a simple worker thread allocator).

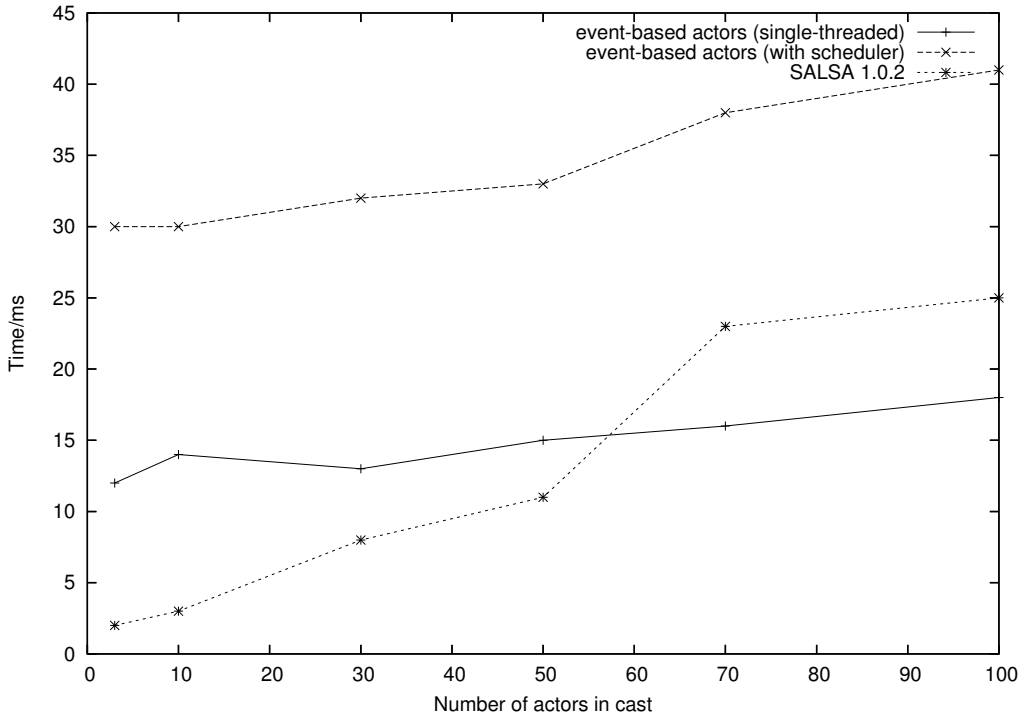

Figure 5.5: Multicast.

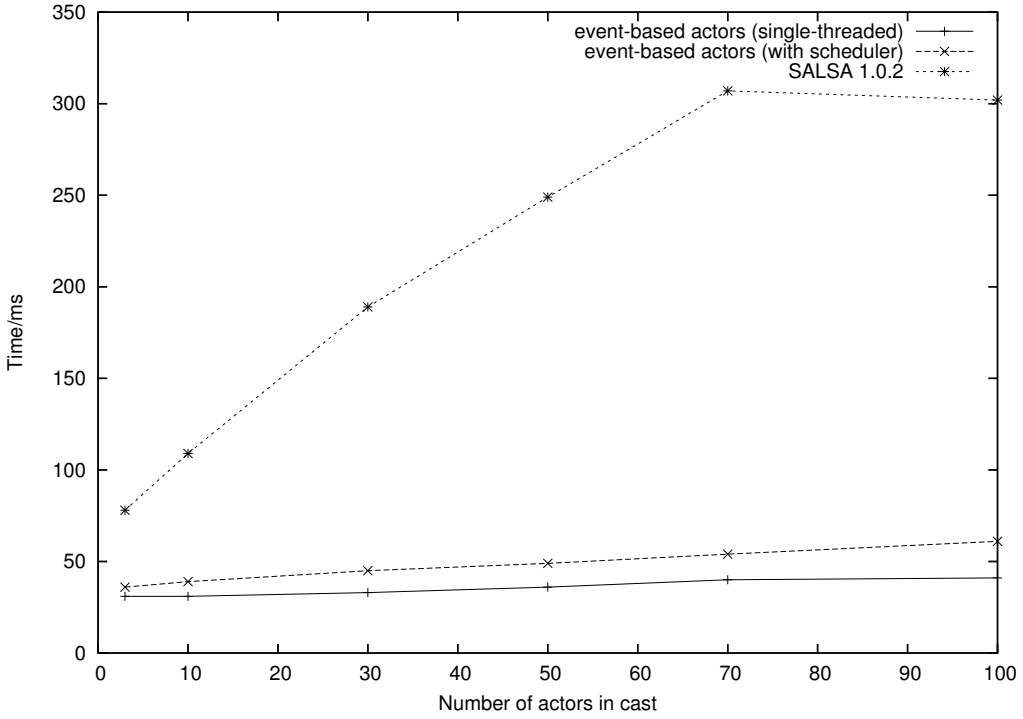

Figure 5.6: Acknowledged multicast.

Scheduling overhead varies between 114% and 128% for 10 or 100 actors, respectively. Note, however, that the single multicast scenario might well be the worst case: As all messages are sent at once, almost none of the worker threads allocated by the scheduler are being reused. Thus, thread creation time dominates the scheduling overhead. For up to 50 actors, SALSA is about twice as fast as single-threaded event-based actors. However, the curves for event-based actors tend to be less steep, testifying to better scalability.

In the acknowledged multicast, depicted in figure 5.6, receipt and processing of a message by the participating actors is acknowledged to the director of the cast. In this case, the scheduling overhead of event-based actors averages 33%. As not all messages are exchanged at once, the scheduler can reuse worker threads.

Group knowledge multicast, depicted in figure 5.7 adds to this interaction an acknowledgment of the director to the participating actors that all actors acknowledged the receipt. As expected, scheduling overhead is even

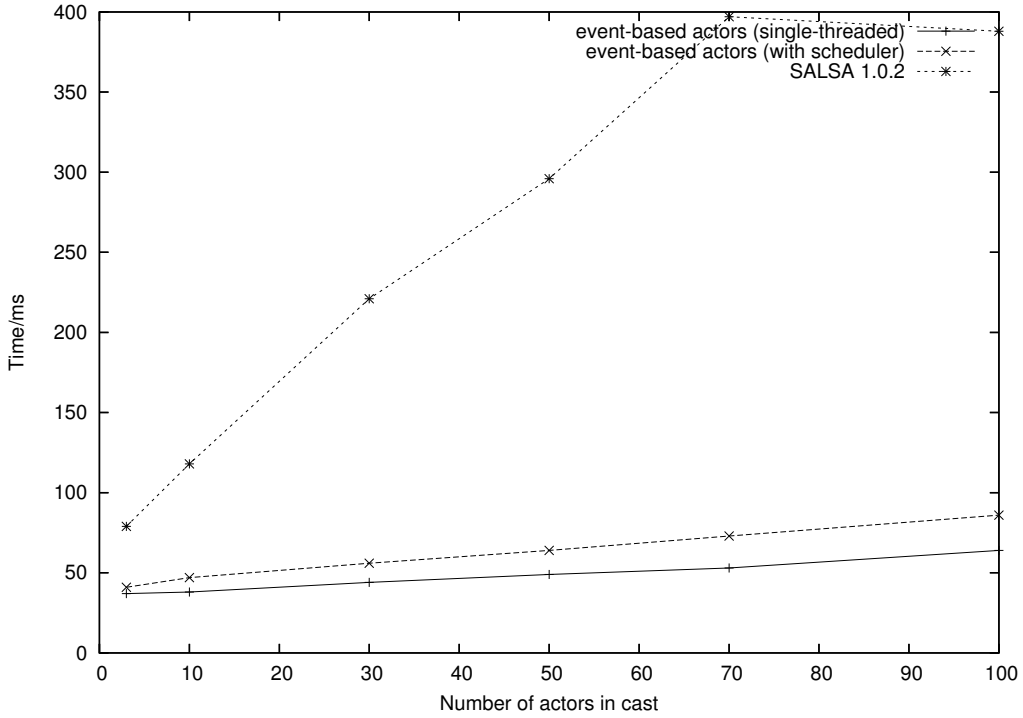

Figure 5.7: Group knowledge multicast.

smaller in this case, and averages about 28%. For both, acknowledged and group knowledge multicast, event-based actors are between 2 and about 5.5 times faster than SALSA.

Performance Summary

We presented experimental results obtained using three different classes of benchmarks. Using a queue-based application we found start-up times of event-based actors to be about 60% slower then Java threads in a worst case scenario. Event-based actors support a number of simultaneously active actors which is two orders of magnitude higher compared to SALSA. Message passing is also much faster: Event-based actors are between 2 and 5.5 times (multicast benchmarks), and over 50 times (Armstrong test) faster than SALSA. Naïve thread-based implementations of our benchmarks perform surprisingly well. However, for high numbers of threads (about 2000 in the Armstrong test), lock contention causes performance to break in. Also, the maximum number of threads is limited due to their memory consumption.

5.2 Case Study

In this section we report on our experiences with implementing a distributed auction service using our event-based actors. In the process we examine typical programming patterns. We will see, that crucial parts of the code are written in a blocking-style code, thereby avoiding the drawbacks of inversion of control.

5.2.1 An Electronic Auction Service

Our electronic auction service mediates between clients interested in selling items and clients interested in purchasing them. For this, a seller creates an instance of an auction actor, sending its PID to interested clients. Clients interested in purchasing items do not interact directly with sellers. Instead, the auction process coordinates all bidding.

Figure 5.8 shows the implementation of the auction actor. The actor's behavior is defined by its run method. Basically, it repeatedly reacts to messages of type `Offer` and `Inquire`, until the auction is closed which is signalled by the special `TIMEOUT()` message. Note that after closing time, the auction stays alive for a period of time, defined by a constant `timeToShutdown`, to reply to late clients that the auction is finished.

At some point, clients interested in purchasing an item are interested in receiving the details of a specific auction which are represented by instances of the following case class:

```
case class AuctionDetails(seller: Pid,
                          descr: String,
                          minBid: int,
                          closingDate: long)
```

Depending on whether the client is interested or not, it may engage into the bidding process. This particular part of the code is depicted in figure 5.9. Note how this entire non-trivial interaction is captured by an intuitive, thread-like sequence of statements. State transitions (e.g. from having received the auction details to waiting for a status message) are explicit in the control flow. Moreover, all state is kept in local variables.

Portability

Currently, there exist two prototype implementations of our runtime system, using TCP sockets and the JXTA peer-to-peer framework, respectively. The presented electronic auction service runs on both platforms. We implemented a simple driver which creates two clients bidding for a sample item. The bidding process can be controlled through a GUI.

In summary, our case study shows that, using our library, non-trivial, distributed actor-based applications can be implemented in a portable manner. Moreover, we showed how typical application-level code can heavily benefit from the intuitive, blocking-style code which is supported by our event-based actors. In an ongoing effort we implement a distributed, fault-tolerant data base as an additional case study. Preliminary results are encouraging.

```
class RemoteAuction(seller: Pid,
                    minBid: int,
                    closing: long) extends RemoteActor {
  val timeToShutdown = 36000000; //msec
  val bidIncrement = 10;
  var maxBid = minBid - bidIncrement;
  var maxBidder: Pid = null;

  override def run: unit = {
    val span = closing - new Date().getTime();
    receiveWithin(span) {
      case Offer(bid, client) =>
        if (bid >= maxBid + bidIncrement) {
          if (maxBid >= minBid) maxBidder ! BeatenOffer(bid);
          maxBid = bid; maxBidder = client; client ! BestOffer;
        } else {
          client ! BeatenOffer(maxBid);
        }
        run

      case Inquire(client) =>
        client ! Status(maxBid);
        run

      case TIMEOUT() =>
        if (maxBid >= minBid) {
          val reply = AuctionConcluded(seller, maxBidder);
          maxBidder ! reply; seller ! reply;
        } else {
          seller ! AuctionFailed();
        }
        receiveWithin(timeToShutdown) {
          case Offer(_, client) => client ! AuctionOver()
          case TIMEOUT() => // do nothing
        }
    }
  }
}
```

Figure 5.8: An auction actor.

```
receive {
  case auction: Pid =>
    receive {
      case AuctionDetails(_, descr, minBid, _) =>
        if (interested(descr, minBid) {
          auction ! Inquire(self)
          receive {
            case Status(maxBid) =>
              makeOffers(maxBid, auction)
          }
        }
        else
          selectNewItem()
    }
}
```

Figure 5.9: Code snippet showing how client actors receive the details of an auction.

Chapter 6

Discussion

6.1 Call/Return Semantics

Calls to `receive` are restricted in that they never return normally. What if an actor wants to return a value after it received a message? Sometimes we are not interested in transmitting explicit information in a return value but merely in synchronous message sends. In this case the receiving actor also needs to send back the usually implicit information that it received the message. Therefore, we collectively refer to the two cases depicted above (synchronous sends returning a value or not) as *call/return semantics* of message sends.

There are basically three ways how call/return semantics can be implemented using non-returning `receives`:

1. Together with the message the sender passes a *continuation function* which contains the remaining computation of the sender. The receiver calls this function at every *return point* in the program. Return values can be passed as arguments to the continuation function.
2. Together with the message the sender passes a *customer*. The passed customer is an actor which is supposed to receive a message which optionally contains return values. Often the customer coincides with the sender. In this case, a send with call/return semantics looks like this:

```
target ! Message(self)
receive {
  case Reply(o) => ...
}
```

3. Return values are passed through *shared memory*, i.e. objects residing on the (shared) heap. For simplicity, we assume that a single value is to be returned through a single object which we call the "container". Sender and receiver have to agree on the container used to pass the return value. For this, the sender may simply pass the container together with the message. Following the send it can retrieve the return value from the container:

   ```
   target ! Message(o); val x = o.get(); ...
   ```

   The receiver simply puts its return value into the container:

   ```
   receive {
     case Message(o) => o.put(retval)
   }
   ```

In summary, the coexistence of actors and objects dramatically simplifies the realization of sends with call/return semantics. The receiver can simply manipulate objects on the shared heap. Customer passing offers call/return semantics in a location-transparent way when *process identifiers* are used as customers. Finally, customer passing tends to be much easier to use than continuation passing, as customers usually provide a more coarse-grained entity and have to be provided only at very specific program points. Also, continuation passing is more difficult to use in a distributed setting as functions have to be serialized instead of process identifiers.

6.2 Event-Loop Abstraction

Our implementation of event-loops given above essentially retains the benefits of Chin and Millstein's approach. On the one hand, the handlers can share state kept in the enclosing scope of their event-loop. Thus, it is easy to ensure that such state is properly manipulated in a *modular* way.

```
def state1 = eventloop {
  case A() =>
    def state2 = eventloop {
      case A() => state3
      case B() => state1
    }
    state2
  case B() =>
    def state3 = eventloop {
      case A() => state1
      case B() => state2
    }
    state3
}
```

Figure 6.1: A finite state machine containing all possible transitions between three states. Reflexive transitions are implicitly provided by the semantics of `eventloop`.

Moreover, state transitions are made explicit by control-flow rather than implicit through updates to data shared among several event handlers.

Moreover, our approach removes some limitations of Chin and Millstein's responders.

1. In their approach, responding methods (the procedural abstraction for responders) are always in a different static scope than the responding block and hence cannot access the responding block's local variables. Therefore, any state needed by a responding method must be explicitly passed as an argument.

   In contrast, our event-loops can be nested. This way, local variables of enclosing event-loops are accessible.

2. Responders, have no support for directly jumping among named event-loops.

   In our case, function (or method) definitions that solely consist of an event-loop can be viewed as named event-loops which permit arbitrary jumps among themselves. As Scala permits nested function definitions, nested event-loops can be named, too. For example,

figure 6.1 shows a finite state machine containing all possible transitions between three states[1].

3. All responding code runs in its own thread. For complex GUIs, such as Eclipse, this approach might pose scalability problems.

   In contrast, leveraging event-based actors, our event-loops are thread-less. Thus, large numbers of fine-grained event-loops can be used.

6.3 Type-Safe Serialization

In the definition of our pickler abstractions we introduced pickler environments. These environments keep track of values pickled (or unpickled) so far.

Pickler environments are passed through existing picklers according to the order in which picklers are applied. This, in turn, is the same order in which bytes are written to the stream. As the construction of our unpicklers is consistent with this order, unpickling functions never have to reverse the list of bytes before unpickling.

The presented combinators are therefore as efficient as those of Kennedy [Ken04]. Kennedy uses circular programming [Bir84] to efficiently thread the pickler state and the stream of bytes in opposite directions. We believe that circular programs should be avoided as they, arguably, make sequential reasoning about program behavior substantially more difficult.

Besides Kennedy's combinator library for Haskell [Ken04], our library is based on a similar library for Standard ML by Elsman [Els04]. In Elsman's library, the pair combinator is primitive, i.e. not defined in terms of a more general combinator, such as the presented sequential combinator. In Kennedy's library, code shared by the `pair`, `wrap`, `list` and `alt` (called `data` in this text) combinators is factored out into the sequential combinator `sequ`.

[1] Note that unhandled events cause the particular event-loops to be executed recursively.

6.4 Limitations

6.4.1 Exception Handling

We want to point out an important limitation of our current exception handling mechanism (see 4.4.2). Updating the state of an actor inside an exception handler should be done *exclusively* using synchronized methods. Currently it is not checked if the actor which installed the exception handler is running at the time of exception handling. Thus, mutual exclusive execution of exception handlers inside an actor is not enforced statically.

We suggest the following solution: Before executing an exception handler, we check if the actor is suspended or not. This can be done by checking if it has a valid continuation set. In this case, the actor is suspended and we can safely execute the handler. To make sure, that the actor is not resumed during exception handling, we temporarily reset its continuation to `null`. This will prevent `send` from scheduling a task item.

If the actor is not suspended at the time an exception is thrown–its continuation is `null`–we queue the exception handler descriptor. The next time the actor blocks, we execute the exception handler as in the case when the actor is found suspended.

6.4.2 Timeouts

We provide a variant of `receive` which can be used to specify periods of time after which a special `TIMEOUT()` message should be received. The mechanism we describe in section 3.5 works well together with our event-based implementation scheme. Checks for timeouts are integrated into the `send` operation and `receive` is unchanged.

However, because the delivery of timeout messages is only triggered when an event occurs (i.e. a new message arrives), timeouts are somewhat imprecise. We provide no estimation or experimental results that characterize their precision. Moreover, we expect timeouts that *generate* events rather than *consume* them, to be much more precise. However, our current

mechanism is suitable to specify *deadlines*. Experience with our case study shows that, sometimes real timeouts can be replaced with deadlines.

Chapter 7

Conclusion

We have shown how lightweight actor abstractions can be implemented on standard, unmodified virtual machines, such as the JVM. For this, we used closures as continuations for actors which are more lightweight than threads.

By enforcing that our blocking `receive` operation never returns normally, dead code is indicated to the user. We found the ability to specify this non-returning property through Scala's type system to be fundamental and important. However, in most mainstream languages, such as Java or C#, the programmer is not able to specify this property in her code, even though it is statically verifiable by the compiler.

By lazily creating threads we were able to guarantee progress even in the presence of arbitrary blocking operations. This technique combined with our event-based implementation yields a unique abstraction. To the best of our knowledge, event-based actors are the first to allow (1) reactive behavior to be expressed without inversion of control, and (2) unrestricted use of blocking operations, at the same time.

However, we found standard virtual machines to be very restrictive with respect to thread handling. For example, on the JVM, (1) operations for suspending and resuming a thread are not dead-lock safe, and (2) user-code cannot find out if a thread is blocked. This makes it very hard to implement efficient concurrency abstractions that are dispatched on multiple threads.

Nonetheless, our actor implementation outperforms other state-of-the-art actor languages with respect to message passing speed and memory consumption by several orders of magnitude. Consequently, a very large number of simultaneously active actors is supported.

We extended our event-based actors with a portable runtime system. The supported programming model closely resembles that of distributed Erlang. Two working prototypes based on TCP and the JXTA peer-to-peer framework, respectively, attest to the portability of our runtime system.

Certain configurations of virtual machines for mobile devices do not support introspection services. Therefore, we implemented a combinator library for type-safe serialization which is as efficient and powerful as other state-of-the-art libraries. As a result, our runtime system is suitable for resource-constrained devices.

All programming abstractions were introduced as a library for Scala rather than as language extensions. We regard four of Scala's language features as essential for the implementation of domain specific languages as libraries: (1) Higher-order functions, (2) lightweight closure syntax, (3) pattern matching, and (4) partial functions. In Scala, these features can be combined to yield unique abstractions that are flexible and easy to use.

7.1 Future Work

First, it would be worthwhile to remove some limitations of our asynchronous exception handling mechanism. We already discussed the mutual exclusive execution of exception handlers and a possible solution in 6.4.1. However, one could go further than that. In the proposed solution the asynchronous exception is queued when the handling actor is found to be busy. Alternatively, a special exception could be thrown inside the handling actor. The exception handler for this special exception would then transfer control directly to the handler of the asynchronous exception. This way, asynchronous exceptions would have a semantics closer to the existing exception handling mechanisms in Java and Scala.

In the near future we are going to cooperate with the authors of the FROB computation model to integrate our two orthogonal approaches to event-

based actors. For this, we plan to modify our event-based implementation to use their low-level Java library. Consequently, we expect to be able to combine the convenience and efficiency of our programming model with the advantages of their resource-aware computing model.

Currently, it is not possible to specify real-time requirements for an actor's behavior. However, our programming model could be modified to allow that. For example, the signature of `receive` could be modified to require certain types of functions as message handlers, say, "actions". Actions would be closures annotated with deadlines or baselines.

Each of those actions would correspond to a task item that is scheduled for execution when a matching message arrives. These task items could be scheduled by a scheduler which implements an appropriate strategy, such as earliest-deadline-first.

Bibliography

[Agh86] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence. The MIT Press, Cambridge, Massachusetts, 1986.

[AHN] Ken Anderson, Tim Hickey, and Peter Norvig. Jscheme.

[Arm96] J. Armstrong. Erlang — a survey of the language and its industrial applications. In *INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan, October 1996.

[Arm97] Joe L. Armstrong. The development of erlang. In *ICFP*, pages 196–203, 1997.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[BCJ+02] A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems, 2002.

[BG99] Legand L. Burge III and K. M. George. JMAS: A Java-based mobile actor system for distributed parallel computation. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 115–129. The USENIX Association, 1999.

[Bir84] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[Bri89] Jean-Pierre Briot. Actalk: A testbed for classifying and designing actor languages in the smalltalk-80 environment. In *ECOOP*, pages 109–129, 1989.

[BSS04] Yannis Bres, Bernard P. Serpette, and Manuel Serrano. Bigloo.NET: compiling scheme to.NET CLR. *Journal of Object Technology*, 3(9):71–94, 2004.

[CM06] Brian Chin and Todd Millstein. Responders: Language support for interactive applications. In *ECOOP*, Nantes, France, July 2006.

[CS05] Georgio Chrysanthakopoulos and Satnam Singh. An asynchronous messaging library for c#. In *Proceedings of Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL), OOPSLA*, 2005.

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *LCN*, pages 455–462, 2004.

[DLL05] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. The jcilk language for multithreaded computing. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, October 2005.

[Els04] Martin Elsman. Type-specialized serialization with sharing. Technical Report TR-2004-43, IT University of Copenhagen, 2004.

[GGH+05] Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. Frugal Mobile Objects. Technical report, EPFL, 2005.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[Gou02] John Gough. *Compiling for the .NET Common Language Runtime*. .NET series. Prentice Hall, 2002.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *ASPLOS*, pages 93–104, 2000.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[Ken04] Andrew Kennedy. Pickler combinators. *J. Funct. Program.*, 14(6):727–739, 2004.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Law02] George Lawton. Moving Java into mobile phones. *Computer*, 35(6):17–20, June 2002.

[LC02] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002. To appear.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[Mat02] Yukihiro Matsumoto. *The Ruby Programming Language*. Addison Wesley Professional, 2002.

[MBC+05] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time java. In *RTSS*, pages 62–71. IEEE Computer Society, 2005.

[NTK03] J. H. Nyström, Philip W. Trinder, and David J. King. Evaluating distributed functional languages for telecommunications software. In Bjarne Däcker and Thomas Arts, editors, *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang, Uppsala, Sweden, August 29, 2003*, pages 1–7. ACM, 2003.

[Oa04] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[PFHV04] F. Pizlo, J. M. Fox, David Holmes, and Jan Vitek. Real-time java scoped memory: Design patterns and semantics. In *ISORC*, pages 101–110. IEEE Computer Society, 2004.

[SMa] Inc. Sun Microsystems. Java platform debugger architecture (jpda).

[SMb] Inc. Sun Microsystems. K virtual machine (kvm).

[SS02] Erik Stenman and Konstantinos Sagonas. On reducing interprocess communication overhead in concurrent programs. pages 58–63, 2002.

[TLD+89] D. A. Thomas, W. R. Lalonde, J. Duimovich, M. Wilson, J. McAffer, and B. Berry. Actra A multitasking/multiprocessing smalltalk. *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, ACM SIGPLAN Notices*, 24(4):87–90, April 1989.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36(12):20–34, 2001.

[Wik94] Claes Wikström. Distributed programming in erlang. In Hoon Hong, editor, *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94 (Hagenberg/Linz, Austria, September 26-28, 1994)*, volume 5 of *Lecture Note Series in Computing*, pages 412–421. World Scientific, Singapore-New Jersey-London-Hong Kong, 1994.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA*, pages 258–268, 1986.

[YT87] Yasuhiko Yokote and Mario Tokoro. Experience and evolution of ConcurrentSmalltalk. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 406–415, December 1987.

Typeset 15th May 2006